

DevSecOps and DevOps for Linux

The Foundations

Marco Antonio Carcano

Apress®

DevSecOps and DevOps for Linux

The Foundations

Marco Antonio Carcano

Apress®

DevSecOps and DevOps for Linux: The Foundations

Marco Antonio Carcano
Genestrerio, Switzerland

ISBN-13 (pbk): 979-8-8688-2076-2 ISBN-13 (electronic): 979-8-8688-2077-9
https://doi.org/10.1007/979-8-8688-2077-9

Copyright © 2026 by Marco Antonio Carcano

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Anandadeep Roy
Editorial Project Manager: Jessica Vakili

Cover image from magnific.com

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To my family, whose unwavering support and strength empower me to face every challenge in life.

Table of Contents

About the Author xxvii

About the Technical Reviewer xxix

Acknowledgments xxxi

Trademark and Open-Source Software Notice xxxiii

Introduction xxxv

Part I: Holistic Skill Set 1

Chapter 1: Team Working 3

Summary 9

Chapter 2: Solution Lifecycle 11

Basic Concepts 12

Solution Lifecycle 15

The Service Design Process 16

The Service Transition Process 21

The Operating Process 24

The Continuous Improvement Process 25

Summary 26

Chapter 3: An Overview of Agile and Lean 27

Agile 28

Lean 30

My Personal Feedback on Agile 30

Define a Scope 31

Project Managers Are Not Solution Architects 34

Summary 34

Chapter 4: An Overview of Scrum and Kanban 35

Scrum 36

Scrum Roles 36

Scrum Project 38

Scrum Sprints 39

Scrum Ceremonies 39

Kanban 41

Kanban Roles 42

Kanban Meetings 44

Summary 46

Part II: Operating with the System 47

Chapter 5: The BASH and Command-Line Utilities 49

What Is a Shell 50

Commonly Used Command-Line Utilities 51

Shell Concepts 56

Commands and Pipelines 56

Subshells 57

The pipefail Option 58

Input and Output Redirection 58

Streams and File Descriptors 59

Here-String 62

Heredoc 62

Xargs 65

Scripting Concepts 66

Execution Permission 67

The Shebang 68

The Special LC_ALL=C Locale 70

Writing Comments 71

Static Analysis Tools 73

Profiling Using the Time Utility 73

BASH Cheat Sheet 74
Script Positional Arguments 74
Variables 75
Globbing 83
List and Sequences 83
User Input 84
Testing Files and Directories 86
Flow Control 86
Control Structures 89
Functions 92
Trapping Signals 95
Summary 96

Part III: Basic Data Formats 97

Chapter 6: Key-Value Pairs 99
Ini 102
Comma-Separated Values (CSV) 106
Summary 111

Part IV: Data Processing 113

Chapter 7: Regular Expressions (RegEx) 115
The Purpose of Regular Expressions 115
Meta-Characters 116
Class of Characters 117
Shorthand 118
Quantification 119
Negating a Class of Characters 122
Logical OR 122
Capturing Group and Backreferences 123
Summary 125

Chapter 8: Working with Grep 127

What Is Grep 127

Scanning Files 128

Scanning All Files in a Directory Tree 129

Case-Insensitive Search 129

Inverting the Matching Criteria 129

Piping to Grep 130

Validating Standard Input 130

Using Grep in Shell Scripts 131

Summary 132

Chapter 9: Working with Sed 133

An Initial Walkthrough 134

Invocation 134

In-place File Modification 136

Record Matching Clauses 138

Showing Only Changes 139

The Pattern Space 140

Hold Space 144

Sed Expression Syntax 145

Nesting Commands 146

Nesting Expressions 147

Using Multiple Expressions 148

Line-Matching Clauses 148

Sed Commands – The Most Used Ones 153

Append (a) 153

Insert (i) 153

Change (c) 154

Delete (d) 154

Substitute (s) 155

Backreferences 156

Summary 157

Chapter 10: Working with AWK 159

Acquainting to AWK 160

Fields Variables 161

Shell Invocation 161

AWK Basic Syntax 162

Pattern-Action Statement 163

Patterns 164

Logical Operators 167

The match( ) Built-In Function 168

Actions 168

Control Statements 169

Variables 171

Declare Variables from the Command Line 171

Declare Variables Within Statements 171

Incrementing and Decrementing Values 172

Modulus 172

Arrays 173

Built-In Variables 174

AWK Functions 176

Generate a Random Number 176

Convert to Lowercase or Uppercase 176

Case-Insensitive Comparison 177

Substring 177

Length of Strings 177

Splitting (Tokenizing) into an Array 178

Sorting an Array 178

Strings Substitution 180

Time Functions 180

Running a Shell Command 181

Custom Functions 182

Real-Life Examples 182
Get the Amount of Space Consumed by Old Files 182
Processes' Memory Usage 183
Summary 186

Part V: Structured Data Processing 187

Chapter 11: XML with XPath, XMLStarlet, and Python 189

What Is XML 190
A Simple XML File 193
Enforcing the XML Syntax with a Schema 195
DTD Schemas 196
XML Schemas 197
Schematrons 197
Example DTD file 198
Example XSD file 200
The XPath Query Language 202
Working with the XPath Command-Line Utility 205
Working with xmlstarlet Command-Line Utility 206
Transforming Using XSLT 210
Fancymonitor with XML support 214
Summary 217

Chapter 12: JSON with JQ and Python 219

What Is JSON 219
Editing JSON with VIM 220
JQ – An Overview 221
JSON Serialization 225
Comments 225
Null Value 225
Scalars 226
Complex Data Types 228

Working with Arrays 230
Number of Elements Using the Length Filter 231
Getting One Element of the List 231
Getting a Slice of the List 231
Sorting a List 232
Getting Min and Max 232
Unpacking a List 232
Getting the Index of an Item of the List with a Certain Value 234
Checking the Existence of a Specific Value 234
More Complex Use Cases 234
Key as Matching Criteria 235
List Item as Matching Criteria 235
Boolean Operators Inside the Select Function 236
Select Items Containing a Specific Key 236
Processing JSON with Python 237
Summary 239

Chapter 13: YAML with Yq and Python 241

What Is YAML 241
Editing YAML with VIM 242
Yq – An Overview 243
YAML Serialization 246
Indentation-Based and Flow-Style Notations 246
Node Types 247
Document Separator 247
Comments 247
null Value 248
Scalars 248
Dictionary 250
List 251
Tags 251
Versions and Schemas 254

Anchors and Aliases 256
Merge Key 258
Validating Using Yamllint 259
Processing YAML with Python 260
A Poorly Written Example 260
Fancymonitor YAML support 262
Summary 264

Part VI: Developing a Python Project 267

Chapter 14: Unit Tests 269
What Are Unit Tests 269
Why Use Unit Tests 269
When to Use Unit Tests 270
Who Writes the Unit Tests 270
The PyTest Framework 270
Install PyTest 271
A Real-Life Example 271
Summary 276

Chapter 15: Packaging As a Wheel 277
The Importance of Packaging 277
Python Packaging 279
Setuptools 279
The Python Build Module 280
Configure the Build Module in an Existing Python Project 281
Including Tests in the Distribution 282
Build a Source Distribution 284
Build a Wheel Distribution 284
Summary 285

Chapter 16: Managing the Build with GNU Make 287

The Importance of a Build Automation Tool 287

What Is GNU Make 288

Installing GNU Make 289

Makefiles 289

A Real-Life Example 290

Summary 292

Chapter 17: Packaging As RPM 295

Platform-Specific Packaging Formats 296

Red Hat Package Manager 296

The RPM Packaging Format 297

Building RPM Packages 298

The Spec File 300

Building the RPM 302

Operating with RPM 304

Summary 307

Chapter 18: Setup Git and the Pre-commit Framework 309

Configure Git 309

Git Commits 310

Commits' Compliance 310

The Pre-commit Hook Python Package 311

Installing the Pre-commit Python Package 311

Configuring the Pre-commit Python Package 311

Upgrading the Scanners 312

Reset the Branch 313

Summary 314

Chapter 19: Auditing Source Code Compliance 315
Static Code Analysis Tools 315
Flake8 316
Pylint 317
Summary 319

Chapter 20: Scanning Vulnerabilities 321
Vulnerability Scanning Tools 321
Bandit 322
Safety 323
Summary 330

Part VII: Security 331

Chapter 21: System Security 333
Files And Directories 335
Date and Time Attributes 335
POSIX Standard Permissions 336
POSIX Special Permissions 340
Extended Attributes 343
POSIX ACLs 345
Mount Options 348
Sudo 350
The Configuration Files 351
The Wheel Group 351
Creating a Simple Sudo Rule 352
Sudo Rule Syntax 354
Linux Capabilities 356
Most Commonly Used Capabilities 356
Getting Capabilities 358
Setting Capabilities 358
SELinux 360
SELinux Policy Sets 360

SELinux Operational Statuses 361
Listing SELinux AVC Violations 363
SELinux Tags 363
Type Enforcement (TE) Rules 365
SELinux Modules 366
Module's Booleans 367
File Context Management 368
Acquainting to Cryptography 369
Symmetric Cryptography 369
Asymmetric Cryptography 372
Summary 374

Chapter 22: Securing Data and Validating Identity Through GPG 377

What Is GNU PG (GPG) 379
The Web of Trust 380
GPG Basic Concepts 381
Primary Keys 381
Subordinate Keys 383
Operating with GPG 384
Listing Keys 384
Exporting and Importing Keys 385
Encrypting And Decrypting Contents 388
Signing Contents 390
GPG-Signed Document 390
Clear-Text Signature 391
Detached Signatures 392
Signing Applications 394
GPG Detached Application's Signatures 394
Package-Specific Embedded GPG Signature 395
Work with GPG-Signed RPM 400
Check if an RPM Package Is GPG Signed 400
Untrusted Signatures 400

Import a GPG Key in the RPM GPG Keyring 400
Valid Signatures 401
Operating the RPM's GPG Keyring 401
GPG Check with YUM/DNF 402
Best Practices for GPG Key Management 404
Summary 404

Chapter 23: X.509 Certificates and PKI 407

X.509 Certificates 408
Self-Signed Certificates 408
Generate the Private Key 409
Generate the Certificate Signing Request (CSR) 410
Generate the Self-Signed Certificate 411
Certificate Validation 412
CA-Signed Certificates 414
Certificate Authority (CA) 415
CA Certificates 415
Public Key Infrastructure (PKI) 415
Generate the Private Key 417
Certificate Signing Request (CSR) 418
Submit the CSR to a Certificate Authority 418
Validating the Certificate 423
Assembling the Certificate Chain 424
Certificate Status 425
Keystores 429
PEM Containers 430
JKS Containers 430
PKCS#12 Containers 432
Private CA 433
Application-Specific Truststores 433
PEM Containers 433

JKS Containers 435
PKCS12 Containers 435
Summary 436

Part VIII: Common Protocols 437

Chapter 24: HTTP 439
The Hypertext Transfer Protocol 439
Request Messages 440
Response Messages 444
HTTP Virtual Host 447
HTTP Proxies 448
Reverse Proxies 448
Outgoing Proxies 449
Summary 451

Chapter 25: TLS 453
Transport Layer Security (TLS) Protocol 453
TLS Handshake 455
Server Name Indication (SNI) 457
TLS Server Certificate Validation 458
TLS Server Certificate Bundle 460
Mutual TLS Authentication 462
Summary 463

Chapter 26: LDAP 465
The Lightweight Directory Access Protocol 465
OpenLDAP Caveats 467
LDAP Schema and Objectclasses 467
Hierarchical Structure 468
Base DN 471
Instantiating Objects 471
LDAP Search Filters 473

Referrals 475
Bind DN 476
The ldapsearch Command-Line Client 476
Other LDAP Command-Line Tools 479
Summary 480

Part IX: Working with Containers 483

Chapter 27: The Containers Ecosystem 485

Containers 485
Container Images 487
Container Image Registries 489
Public Online Container Registries 490
Private Online Container Registries 491
On-Premises Container Registries 491
Summary 492

Chapter 28: Docker and Podman 493

Container Management Suites 494
Docker 495
Podman 496
Operating Container and Images 498
Operating with Container Image Registries 498
Running Containers 509
Creating Container Images 518
Docker 521
Podman (buildah) 521
Networking and Volumes 522
Cleanup 524
Summary 524

Chapter 29: Deploying Pulp3 Using Docker Compose 525
Docker and Podman Compose 525
Example Deployment: Pulp3 526
What Is Pulp 526
Deploy Pulp3 527
Log In to the Pulp3 Web UI 530
Install the Pulp CLI 531
Create a Repository in Pulp3 531
Summary 532

Part X: Service Protocols 535

Chapter 30: SOAP and REST 537
Simple Object Access Protocol (SOAP) 537
Web Services Description Language (WSDL) 539
Mocking a SOAP Endpoint 539
Representational State Transfer (REST) 543
OpenAPI 544
Mocking a REST Endpoint 546
Summary 549

Part XI: Orchestrating Containers 551

Chapter 31: Kubernetes, Helm, and Rancher 553
Kubernetes 554
Kubeadm 554
Minkube and K3s 554
Openshift and Rancher 555
A Quick Walkthrough with Kubernetes 555
Namespaces 556
The Kubectl Command-Line Tool 556
Resource Definitions 559
Custom Resource Definitions and Operators 569

Helm 570
Install Helm 571
Add Repositories 571
Search Helm Charts 572
Pull Helm Chart's Contents 572
Deploy Using Helm Charts 573
Rancher 576
Deploy the Rancher Single Node Docker Container 577
Prerequisites 578
Deploy Rancher 580
Operating K3s 582
Connection to Rancher 583
Rancher Backups 587
Active Directory Authentication 592
Summary 596

Part XII: CI/CD Toolchain 597

Chapter 32: Deploying Gitea Using the Helm Chart 599

What Is Gitea 600
Deploy Gitea Using the Official HELM Chart 600
Requisites 600
Gitea Helm Charts 606
Deploy Gitea 608
Deploy Gitea Actions Runners 615
Generate an Actions Runner Registration Token 615
Store the Token into a Kubernetes Secret 616
Deploy the Gitea Runner 617
Verify the Gitea Runner Registration 618
A Quick Walkthrough with Gitea 619
Create an Organization 619
Create a Repository 620
Create an Actions Workflow 621

Gitea Helm Chart Caveats and Pitfalls 623
Configure the Domain 623
Configure Webhooks 624
Configure Migration Sources 624
Gitea Version 625
Load Balancers 625
TLS Certificate 626
Actions Runners with TLS-Enabled Gitea Endpoint 628
Actions Runner Tweaks 629
Configuring Logging 630
LDAP Authentication 631
Enable Monitoring 633
Configure Resource Requests and Limits 634
Scaling 635
Summary 637

Chapter 33: CI/CD and Continuous Delivery 639

CI/CD 640
The CI/CD Toolchain 641
Source Code Management (SCM) 641
Repository Managers 642
Build Automation Facilities 645
Automated Testing Facilities 646
Continuous Integration (CI) 647
Development 647
Continuous Building 648
Continuous Testing 649
Continuous Deployment (CD) 650
Deploying to the Staging Environment 651
Deploying to Customer-Facing Repositories 652
Continuous Delivery 653
Summary 655

Chapter 34: Working with Gitea Actions Workflows 657

What Are Gitea Actions Workflows 658

Pushing Fancymonitor to Gitea 659

Create the Fancymonitor Repository in Gitea 659

Add the Remote Origin to Fancymonitor 661

Pushing the Main Branch to the Remote Origin 662

The Unit Test Workflows 663

Mocking Services 666

Create the Mock Container 667

Configure the Actions Workflow 668

Pull Requests 671

Pull Request's Merging Strategies 671

Pushing a Branch 671

Raising a Pull Request 672

Completing a Pull Request 674

Creating a Release 675

Configuring the Push Target 676

Triggering the Create Release Workflow 679

Summary 682

Part XIII: Version Control 685

Chapter 35: Version Management with Git 687

Acquainting to Git 688

Git Concepts 689

Areas 689

Branches 690

Tags 691

Install Git 691

Configure Git 692

Working with Git Repositories 693

Creating a New Repository 693

Linking the Local Git Repository to a Shared Remote 695
Set the Upstream of the Current Branch 695
Cloning an Existing Repository 696
Version Control Operations 697
Populating the Working Area 698
Staging Modified Contents 699
Checking Differences 700
Unstaging Contents 701
Committing Contents 703
Amending a Commit 703
Listing Changes in the Current Branch 704
Reverting a Specific Commit 705
Rewind to an Earlier Point in History 706
Comparing Commits 708
Remote Repositories 710
Cloning a Remote Repository 710
Listing Remote Repositories (Remotes) 711
Linking to a Remote Repository 712
Working with Branches 712
Check Out a Branch and Managing Branches 713
Stashing 716
Remote Branches 721
Merging Branches 728
Branch Merging Strategies 729
Revert a Merge 745
Resolve Conflicts 747
Tagging 750
Creating a Tag 751
Listing Tags 751
Deleting a Tag 752
Checking Out a Tag 752

Remote Tags 752
Hooks 754
Summary 755

Part XIV: Remote Configuration Management 757

Chapter 36: An Overview of SSH 759

SSH Basics 760
Host Keys 761
Host Key Fingerprint 762
Getting the Host Key Fingerprint 763
Authentication Mechanisms 764
Password-Based Authentication 764
Public Key Authentication 764
Client Configuration Files 770
Connecting to a Server 772
Executing Remote Commands 773
Secure Copy – SCP 774
SFTP 776
A Few SSH Server Security Best Practices 776
Summary 779

Chapter 37: Ansible Playbooks, Roles, and Collections 781

Containerized Ansible 782
Set Up the Container Environment 783
Pull the Ansible Container 783
Set Up the Ansible Directory Tree 783
Configure Ansible 783
Run Ansible Within the Container 786
Launching the Containerized Ansible Directly from the Shell 788
Ansible Ad Hoc Statements 790
Ansible Playbooks 793
Writing an Ansible Playbook 794

Running a Playbook 797
Ansible Namespaces 798
Ansible Roles and Collections 799
Ansible Roles 799
Ansible Collections 800
Ansible Galaxy 800
Working with Ansible Roles 801
Install an Ansible Role 801
Write a Playbook Invoking the Role 801
Write a Var File 803
Run a Play 804
Ansible Vault 806
Create a Vault File 806
Display the Contents of a Vault File 807
Edit an Existing Vault File 807
Use a Vault File 807
Work with Ansible Collections 808
Install an Ansible Collection 808
Invoke Modules from a Collection into a Playbook 809
Run a Play 810
Structuring the Playbooks Directory Tree 810
The Infrastructural Playbooks Directory Tree 811
The Solutions Playbooks Directory Tree 813
Summary 814
Conclusion 815

Index 819

About the Author



Marco Antonio Carcano is a long-time Linux professional based in Tessin, Switzerland. He is a seasoned Linux professional with extensive experience in the field. Beginning his career as a developer, he later transitioned into systems roles, primarily within the financial sector, where he has held positions such as Systems Engineer, DevOps Engineer, Systems Architect, and Solutions Architect.

A strong advocate for automation, continuous integration, and continuous delivery, he emphasizes the importance of infrastructure as code to enable repeatable, scalable, and reliable deployments. He champions the use of rapid feedback loops – from automated testing, monitoring, and user insights – to drive continuous improvement and ensure solutions evolve to meet changing needs.

He believes that solutions should not only function effectively but also prioritize delivering the best possible user experience. To achieve this, he promotes the adoption of standards and best practices – not only in the design and management of solutions but also in implementing effective governance and embedding security throughout the development lifecycle. Reflecting core DevSecOps principles, he fosters a culture of collaboration, shared responsibility, and continuous learning across development, operations, and security teams. He believes that this culture must be supported by strong governance and a commitment to comprehensive documentation, ensuring maintainability, transparency, and knowledge sharing.

He is convinced that this holistic approach – delivering reliable, secure, and continuously improving solutions in challenging and sensitive environments – benefits the entire organization by generating real value with minimal waste and risk. Additionally, this strategy enhances the quality of life for those working in such settings by reducing stress, eliminating unnecessary internal conflicts, lowering turnover, and, most importantly, helping the company retain its most valuable assets: employees' knowledge, skills, and expertise.

If you fancy, you can follow him on LinkedIn at https://www.linkedin.com/in/carcano/.

About the Technical Reviewer



Maksim Avilov is a software developer and team leader with more than 25 years of experience in software development. He focuses on modernization of legacy systems, Agile delivery, and DevOps practices. Over the years, he has led diverse co-located as well as distributed teams in building effective software solutions, while fostering collaborative, high-performing environments and even mentoring apprentices. Originally from Russia, he is based in Zurich, Switzerland, where he enjoys running and cycling in the Alps. Connect with him on LinkedIn: linkedin.com/in/avilov.

In reviewing this book, Maksim brought his expertise across a broad spectrum of technologies and practices, ensuring that examples and labs are both technically sound and practically relevant. He is passionate about helping teams adopt modern approaches to software delivery and bridging the gap between established systems and contemporary engineering practices.

Note from the Technical Reviewer

When Marco first approached me about reviewing *DevSecOps and DevOps for Linux*, I wasn't entirely sure I could fit it into my schedule. After giving it some thought, I quickly realized this was a challenge worth accepting. The book spans an incredible range of topics, from core Linux concepts to advanced DevSecOps practices, and each chapter came with its own examples, labs, and details to carefully verify. This was an opportunity for me to look at these subjects from Marco's perspective.

What I enjoyed most was not just checking the technical accuracy but also seeing how the content fits together to guide the reader step by step. It reminded me why I enjoy being part of the software community: helping ideas become clearer, sharper, and more useful to others.

With this foundational knowledge, readers must be comfortable navigating the complexity of existing and prospective self-hosted and cloud DevSecOps platforms and should be prepared to tackle professional challenges in a modern fast-paced environment.

I'm grateful to have had the chance to contribute to this project, and I hope my work as a reviewer makes your reading journey smoother and more rewarding.

Maksim Avilov

Acknowledgments

Dear reader, this book distills what I consider the foundational pillars of DevOps and DevSecOps on Linux, drawing from my professional experience. Writing it was a formidable challenge: DevSecOps is a vast field that inherently includes DevOps, requiring careful curation to fit within the constraints of a single volume.

Even just selecting essential topics demanded rigorous prioritization: I aimed to appeal to both students and seasoned professionals adopting DevSecOps (and DevOps by extension). Ultimately, I set a baseline of basic Linux skills, including shell and Python scripting – this should be readily accessible to vocational or college students.

I wanted to craft an engaging style: in the hope to make the content compelling, I blended high-level theoretical concepts with in-depth dives, supported by realistic hands-on labs. These labs extend beyond scripts to include tools and environments typical in DevOps and DevSecOps workflows.

And this was another huge challenge – designing the lab environment. This really was another hurdle, since I wanted to create a practical lab setup which minimizes system disruption and resource demands, even on quite modest hardware. I resolved this by consolidating everything into a single virtual machine, integrating Rancher, Kubernetes (via K3s), and deployed services.

Writing this book was certainly one of the toughest projects of my career: the complexity and number of topics required me to keep very focused for nearly one year, sacrificing personal time, but the result justified the effort.

I would like to extend my heartfelt thanks to Maksim Avilov for accepting the challenge of technically reviewing this book on such short notice. The breadth of topics covered is extensive, and the number of examples and labs he reviewed was substantial, encompassing a wide range of diverse subjects. It is not easy to find a trusted professional capable of revising such a vast array of content, and I truly appreciate his expertise and commitment.

Lastly, but not surprisingly, I owe a huge thanks to my family too. A special thanks to my wife and sons, for supporting me and tolerating my long-term mental absence, and to my two cats Romeo and Anubi, who tolerated me staying in their kingdom's territory while writing.

Trademark and Open-Source Software Notice

All trademarks, service marks, trade names, and product names appearing in this book are the property of their respective owners. The practical examples and laboratories illustrated throughout this work utilize various open-source software, packages, platforms, and interactive environments.

Except for the author's original code, scripts, and configurations, these tools are utilized under their respective open-source licenses (including but not limited to Apache 2.0, GPL, MIT, and BSD). All credit for the underlying source code, platform architectures, infrastructure, and official documentation belongs entirely to their original authors, contributors, and maintainers. The technologies featured herein are used strictly for instructional and educational purposes.

Introduction

This book, crafted to meet the needs of both college students and experienced Linux professionals willing to quickly acquire the pillars of DevOps and DevSecOps on Linux, is a comprehensive guide tailored to help you develop the core skills necessary to succeed in this rapidly evolving field.

DevOps and DevSecOps represent challenging disciplines, since DevOps and DevSecOps professionals find themselves right in the middle of development, quality assurance (QA), and security teams.

The strictest interpretation of DevSecOps involves professionals facilitating communication and processes across teams, ensuring that security is an integral part of the development process and that every solution is delivered with security considerations addressed throughout. In my opinion, this strict interpretation may be overly limiting.

The widest interpretation of DevSecOps encompasses DevOps, as it builds on the collaborative and efficient practices of DevOps while adding a critical focus on security throughout the development lifecycle. This interpretation assumes that DevSecOps professionals are also familiar with DevOps, which is essential for effective collaboration and integration of security into DevOps processes.

This book focuses on the widest interpretation of DevSecOps, which blends security concepts into DevOps. For this reason, it illustrates not only concepts strictly related to DevSecOps, but also DevOps concepts since it is not correct to take them for granted.

Learning the widest interpretation of DevSecOps will enable you to actively support development, QA, and security teams. You will be equipped to develop secure CI/CD (continuous integration/continuous deployment) integrations and automations, identify and resolve security threats, and facilitate communication and problem-solving across these teams.

The baseline skillset for reading the book is

- Being able to autonomously install a Linux system, preferably into a virtual machine (VM)
- Basic understanding of LAN networking

- Basic understanding of the Linux shell, including very basic scripting
- Basic understanding of Python scripting
- Basic usage of the VIM editor
- Basic understanding of SSH

In addition to the necessary theory, the book provides lots of examples including hands-on labs.

To stay focused on the described solutions, avoiding the overhead and distraction of describing every time the nuances of different Linux distributions, it has been chosen to use only one reference platform: Oracle Linux 9.

The examples are supposed to work also on other Red Hat Linux 9 family distributions, with minor modifications as necessary.

Although the examples have been thoroughly tested and working at the time of writing this book, it is obviously impossible to guarantee to have all of them fully operational after the publishing date, since the involved software may introduce incompatible changes in the meantime.

If the reader wants to use another Linux distribution or version, it's up to them to rework the examples to have them working on the chosen platform.

Note This book condenses a vast array of complex topics and concepts. Except for the initial chapters and a few other sections, it is challenging to fully comprehend the material without actively running the examples and completing the labs. In addition to that, the examples and labs have been tailored to provide an engaging learning experience gradually improving the reader's knowledge adding the new concepts only after introducing their prerequisites: the side back of this approach is that **it is not possible to skip examples and labs and focus only on specific ones – examples and labs must be done and completed in the exact order they have been put in this book**.

To avoid messing up systems, the labs and examples have been tailored to be run on a single machine (preferably virtual).

Note This book's labs, examples, and material are designed to illustrate the concepts discussed and serve only as educational references. They are provided without any guarantees, either implicit or explicit. Users must run the examples, labs, and any provided material on a dedicated and isolated machine or virtual machine (VM) intended as a lab, within a controlled environment that has Internet access, taking all necessary precautions to prevent interference with other systems, especially production ones. Due to the integration of various software and solutions, the compatibility and functionality of the labs may change over time as each project evolves. Where possible, the software versions used in the labs are specified to provide a tested reference. However, updates to third-party software may still cause compatibility issues which may also lead to breaking the lab or examples: since these are unpredictable events, resolving these issues is the user's responsibility. Some labs include third-party software with licenses provided in the companion resources. The author and publisher disclaim any liability for damages or losses resulting from the use of this book's examples, labs, and materials. By using this book's examples, labs, and materials, you acknowledge and accept these risks and agree to take full responsibility for any consequences.

Since some of the labs require running a stack of Docker with K3s and Rancher on top of it, the minimum requirements are a little bit demanding:

- **Linux distribution**: Red Hat Linux 9 family (Oracle Linux, Rocky Linux, etc.).
- **Cores**: Four (strongly suggesting at least six).
- **RAM**: 16 GiB.
- **Hard disk**: 100 GiB.
- **Partitioning**: You can use a partition scheme of your choice, but it is mandatory to have at least 60 GiB for the partition where the `/var/lib` directory tree is stored.
- Internet connection.

The examples and labs have been tested on

- Virtual machine with 16 GiB of RAM and six cores on an Intel Core i7 3600 MHz running Rocky Linux 9 (official Vagrant image version 6.0.0)
- Virtual machine with 16 GiB of RAM and six cores on an Apple M1 Max running Oracle Linux 9

For your convenience, the book's supplementary material provides a `Vagrantfile` for easily provisioning an Oracle Linux 9 VM with the correct requirements. Instructions for retrieving this book's supplementary material are available at the very beginning of the book.

If you want to use it, you must first install `Vagrant` and `Virtualbox` – the installation process is trivial, so the detailed installation instructions are not provided in this book. If you need some guidelines, you may find it useful reading my blog post https://grimoire.carcano.ch/blog/vagrant-installing-and-operating/.

Once `Virtualbox` and `Vagrant` are installed, you still need to fetch the Oracle Linux 9 Vagrant box for your hardware architecture.

Luckily, Oracle provides official Vagrant boxes: they are available at https://yum.oracle.com/boxes/.

To download the one you need, proceed as follows.

First, set the base URL from where the boxes are available for download into the `BOXES` environment variable:

```
BOXES=https://oracle.github.io/vagrant-projects/boxes
```

Then, if you are on a `x86_64` platform, run

```
vagrant box add --provider virtualbox \
${BOXES}/oraclelinux/9.json
```

If instead your platform is `ARM`, run

```
vagrant box add --provider virtualbox \
${BOXES}/oraclelinux/9-aarch64.json
```

Both statements download a copy of the official Vagrant box: just wait for the process to complete.

Once done, create a directory to be used as the workspace for the Vagrant deployment (e.g., `devsecops-book`), copy the provided `Vagrantfile` into it, and change the directory to it.

Note The provided `Vagrantfile` supports shared folders: more specifically, it configures the automatic mount of the `/vagrant` directory in the Vagrant VM to the current working directory of the host running it. At the time of writing this book, the Vagrant boxes provided by Oracle support shared folders. Since this feature is tightly bound to the used kernel, upgrades may occasionally break it. If you are facing problems, you can disable it by setting `disabled` to `true` in the `config.vm.synced_folder '.', '/vagrant', disabled: false` statement in the `Vagrantfile`.

Before going further, you must also copy in the Vagrant's deployment directory (`devsecops-book` in this example) the `devsecops-pillars.tar.gz` containing the book's companion resources – this tarball is part of the supplementary material: you can retrieve it the same way you did for the `Vagrant` file. Just copy the file to the same directory where you put the `Vagrantfile`, but don't extract it so far.

You can now spin up the lab's VM – just type

```
vagrant up devsecops_lab_01
```

Note This `Vagrantfile` contains the necessary logic to automatically install any missing required Vagrant plugins. If a plugin is installed during the execution of the above command, the process will terminate immediately after the installation completes. In that case, simply re-run it and repeat as needed until all required plugins are installed. Mind that it also performs a check to automatically select the vagrant box version depending on your architecture. If you get the error `Vagrantfile:33:in `block in <top (required)>': Unsupported architecture`, you can add your architecture to line `24` if your machine is an `x86_64` or to line `28` if your machine is an `AARM`. Mind also that the Vagrant VM needs an Internet connection – if it is not available, it hangs in the inline script section.

Depending on your host's setup, the provisioning process may probably hang with the following question:

```
==> devsecops_lab_01: Available bridged network interfaces:

1) en0: Wi-Fi
2) en1: Ethernet Adapter (en1)

devsecops_lab_01: Which interface should the network bridge to?
```

The correct answer depends on your setup: you must specify the number of the network interface with Internet access.

Supposing it is the Wi-Fi network, then answer `1`.

If you want to avoid being asked this question each time you run the `vagrant up` statement, please modify in the `Vagrantfile` the value of the bridge setting in the line starting with

```
node.vm.network "public_network", bridge: "en0: Ethernet"
```

so that it matches your actual setup.

Wait for the provisioning process to complete, then login to the VM by running

```
vagrant ssh devsecops_lab_01
```

You can now operate the VM as needed.

To disconnect from the VM and get back to the host's shell, just type `exit` or press `CTRL+D` as you would usually do to terminate SSH sessions.

To shut down the lab's VM, on the virtualization host, just type

```
vagrant halt devsecops_lab_01
```

Now, every time you need to use the VM again, run `vagrant up devsecops_lab_01` to start it, followed by `vagrant ssh devsecops_lab_01` to connect to it.

If you messed up the lab's VM, you could destroy the lab's VM and provision it as a new one – **of course this causes losing all of the VM's current contents**:

```
vagrant destroy -f devsecops_lab_01
vagrant up devsecops_lab_01
```

PART I

Holistic Skill Set

DevSecOps builds directly upon DevOps, uniting technical practices with workflow structures that mandate collaboration across teams. Consequently, practitioners require the precise communication skills necessary to align development, security, and operations on unified security standards.

Because these responsibilities span the entire software lifecycle, operations must integrate fluidly with every department.

Furthermore, since DevOps and DevSecOps professionals drive modern project delivery, mastering standard management frameworks is mandatory to ensure both velocity and compliance.

High-value engineers must therefore complement their technical expertise with a comprehensive understanding of the solution lifecycle and agile methodologies like Scrum, Lean, and Kanban.

The chapters in this section provide this operational foundation, turning core engineering skills into high-impact professional value.

CHAPTER 1

Team Working

As anticipated, DevOps and DevSecOps emphasize collaboration between teams. Believe me or not, the root causes of a lot of the problems you find in every workplace are not technical: systems are governed by humans, and if you follow the lines to their root, you will see that most of the failures, when they are not caused by too small budgets, are nearly always bound to human-related shortcomings, such as too poorly designed processes, too strict deadlines, and poor team working capacity.

Teamwork is one of the most difficult soft skills to develop, but it is invaluable – you can be the most valuable professional in your company, but it is not worth much if people don't like to work with you. In addition to that, you risk becoming part of the problems your colleagues must face daily.

The first golden rule comes from a note which was attached to the door of the recording studios when they were recording the charity single "We Are the World" by the USA for Africa supergroup:

> *Check your ego out at the door*
>
> – USA for Africa

Teams are made up of different people, each one with their own habits, culture, beliefs, past experiences, and personal life (and of course personal problems): thinking you are the best and others are worth less than you doesn't certainly help you to get along with others. Always remember the DevSecOps and DevOps job is facilitating things and sorting out problems: having to sort out a team's internal problems caused by misbehaving people is really the last thing anybody wants to deal with, so prima donnas are never welcome.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_1

Remember you are human

Sometimes on LinkedIn I see weird posts teaching ways to avoid saying "sorry", … and even worse I see positive likes and congratulating comments of people thinking that this attitude is good. It is very bad indeed: we are human beings, human beings make errors, and polite humans say sorry when they make a mistake. Avoiding saying sorry after having made an error just makes others think you are either an idiot because you don't recognize your mistake or just a megalomaniac thinking to be some kind of infallible God and others worth nothing. My personal advice, if the problem you caused affected someone, is to always apologize and, even better, to show you are really sorry, to offer them a drink or a coffee if it is possible.

Always carefully think about your colleague's hints

You will receive hints from your colleagues, sometimes even unsolicited ones: give the same initial worth to everyone's hints, even if they come from people having far less working experience than you – you will be pleasantly surprised how sometimes on specific topics anybody can have more experience than you or simply got something you missed.

Always help your teammates

Always help anybody asking for it: they'll certainly be grateful and it's very likely that they will return the favor when you will be the one in need of help. If they don't ask for help but you see them in trouble, then it is a little bit trickier, since some people don't like to be helped. In my experience a good way to go is to use indirect questions to figure out the problem and, if you feel you know how to address it, give them just the hints necessary to find the solution themselves – they will be grateful to you for your hints, yet be proud of having found the solution themselves.

Be self-confident, yet aware of your own limits

Our goal is quickly providing values to the corporate, not to demonstrate to others that we are the champions, so don't be too proud neither too shy – when necessary, always seek your teammates' help: maybe their different experiences can help you to avoid design or setup errors or to quickly sort out complex problems you see for the first time. Your company will benefit from the faster solutions; you will benefit from their help, being less frustrated and stressed. Additionally, you will help them develop their

self-esteem since they will see you understand their value. It's a win-win, where you both are improving your quality of life. When you have been helped, while talking with others about what you did, never forget to mention who helped you.

Try to (safely) overcome your own limits

I said safely, which means to spare ourselves from unmanageable risks: learn concepts and processes from already skilled people and do not forget to keep yourself current with the bleeding-edge technologies (at least the theory). Improving your skills is always a winning choice but mind I'm not saying the new is always good – always use your own judgment to guess if changing something is really worth the effort, carefully considering not only the provided value, but also any drawbacks – changing to anything new only because it is the more modern way fitting a use case may not be a wise choice: sometimes it's best to keep what you already have.

Celebrate achievements and share your credits with the rest of the team

Achievement must always be celebrated, no matter if it is your own or that of your teammates. In addition to that, always advertise your own achievements as team achievements and not just yours: from the perspective of the top-level management, success or failure is perceived at a team level, not at a personal level. Sharing your credits with the team will also strengthen the confidence your teammates have in you, since they will see you as someone who they can trust and who really cares about the team. And when talking about your success, learn to say "we" in place of "I."

Note Some odd team leaders think that putting everyone in competition pushes people to do their best. I totally disagree with this archaic and dumb approach: the only outcome is in the best case having everybody working for himself fighting colleagues, with the risk in the worst case to have somebody intentionally causing issues to others trying to emerge – it's the same as in a football team where the coach encourages each player to play the game alone, considering each one as an enemy – the other team will clearly easily win against them. If during a job interview you suspect you are dealing with someone like that, my suggestion is to quit the interview immediately – that working environment is certainly toxic.

Communicate in an excellent way

DevSecOps and DevOps practitioners are right in the middle of everything: it is mandatory being very good at communicating, in the cleanest and most professional possible way, using the proper terminology, avoiding taking things for granted when describing things yet avoiding unnecessary details. Same way as in designing, also in communication the best way to go is the *KISS* (*Keep It Simple, Stupid*) approach. Too many times, I have seen poorly written Service Requests and Incident Reports lacking not only basic but even critical information to figure out the proper context: the point is that, when working at big corporations, people of different teams have no clue about other teams' jobs. The consequence is that the support teams always need to get enough context to be able to start working on a request or to properly figure out a solution. So, mind this: *the more detailed the request is, the less it will take to be fulfilled* – otherwise a thread of comments asking for more context will be started, and until you provide all the necessary information, your request won't start processing. Please remember also not to provide unnecessary details: it's a waste of your time, a delay in submitting the request, and provides no benefit in solving your issue.

As for how to communicate, always prefer written communications: they are much easier to understand, and they can be re-read as many times as necessary. Emails are very good when dealing with long messages, generating threads that are expected to be resolved within a few replies or in a short amount of time. Instant messaging is most suitable for requests you think can be sorted out with a few short answers or when you need an immediate answer. Sadly, I saw a lot of troubles caused by people not being good at communicating, no matter how professionally skilled they are: in the best case they just cause unnecessary delays and frustration and stress for the whole team, but in the worst, they can even cause service disruptions and money loss.

Before submitting Incident Reports, always perform an initial analysis

Opening Incident Reports to the wrong team causes delay to the solution and is a waste of resources and energy. Sadly several times I saw Incident Reports opened to the wrong support teams: this can happen when not enough or wrong information are provided to perform a proper triage or when a second- or third-level support resource finds a problem outside its working scope – in this case the triage is often skipped, since the requestor is supposed to be skilled enough to conduct the triage himself.

As an example, consider you are working as a middleware operations engineer and you find an entry in a log file complaining about not being able to connect to a database:

your first reaction probably would be to raise an Incident Report to the database administrators team, … but the root problem can also be an invalid DNS configuration, an expired X.509 certificate, a mismatching firewall rule, an intrusion prevention system gone out of control, or any other kind of networking problem. So, the root cause can be in the scope of three other different teams: the one governing the DNS servers, the identity management team, and the networking team. Immediately raising the request to the database administrators team risks just delaying the solution of the problem and creating unnecessary chaos, besides damaging your own reputation, since you were not good at making a simple triage.

Debate and share knowledge and skills

Everyone in the team benefits and appreciates team members sharing knowledge and skills: this attitude improves both team productivity and team members' mutual respect. Why reinvent the wheel? It is much better to benefit from those who already did it and live less frustrated and stressed. Of course, if you can improve it as necessary, do it, but remember to keep the change focused on your task and to share your findings. Mind also not to change too much the wheel, … otherwise you are reinventing it – if you really feel it's the case, talk with your team before doing that. If you see oddities, try not to underestimate your colleague's previous work: maybe they were firefighting or on a too strict milestone; for sure they had a good reason for doing something like that, so don't judge and, before improving, let them know what you are doing.

Use only officially corporate-approved documentation software and services for documenting your findings and try to structure posts and articles in a straightforward way so that they are easy to find by your colleagues.

Don't call unnecessary meetings

One of the pain points of many companies nowadays is wasting a lot of time because of unnecessary meetings on topics that can be easily solved with a few emails. Always try to get your answers and address things by using written communications first: if after a few messages you don't agree on some topics, you can always call a meeting and discuss only the mismatches. This approach will save a lot of your and your colleagues' time, besides giving you also the convenience of a tracked record that you can check if you forget things after time passes by.

Limit recipients in communications

Unneeded communications are just background noise in people's work – they are like spam, and they have all the side effects of spam indeed. As an example, let's suppose that you receive a lot of off-topic (for you) emails on a project you are working on: reading them to make sure they don't really require your support will quickly become a pointless and frustrating activity which subtracts too much time from your real work. As a consequence, you will start looking at them less frequently, skimming contents instead of reading. In the worst case, you will probably even create mail filters to address them to the right place (the trash bin!).

This of course increases the risk you are missing important information that is hidden in the middle of that bunch of pointless information.

Don't be like these chaos emissaries: since they are often causing problems, these spamming guys are also very often not very popular with the other co-workers. Always take some time to scope which people or teams are really recipients of the specific topics you are writing about – as an example, consider a project where there are architects, DevOps, and developers: an email concerning a configuration to apply for a specific system infrastructure certainly is worth nothing for the developers, … sending it to them is just sending spam.

Don't blindly execute tasks be smart and proactive

Always carefully think about what you are doing, not only in the strict technical field: tasks and requests often hide pitfalls that can cause troubles to your team, to other teams, and in general to the business. Consider this real-life experience I had: some time ago I was told to set up the notification system of an application. The (dumb) answer to my question about which recipient to set up was to send everything to the DevOps team. If I blindly did what I was told, I would have caused a lot of problems: that notification system does not only send technical alerts, but also business notifications, which are supposed to go to the backoffice, so they can quickly address customer needs. Since I am diligent, I spent a couple of hours investigating how that notification system works and I figured out that the proper setup requires three different recipients: one for technical alerts (so DevOps team was the correct one), but the other two were for business notifications (one was for customer request; the other was for administrative tasks related to business operators). My diligence not only prevented the spreading of problems and chaos: it also provided benefits. The DevOps team was not being overwhelmed by receiving meaningless (to them) notifications, receiving instead

only technical alerts relevant to them. The backoffice was able to receive immediate notifications about customer requests, and any application-specific administrative tasks were immediately notified to business-level administrative users. It was a win for everybody and the defeat of the chaos, meaning quick response for both business and technical problems. Guess how long a business request can be delayed if it is notified to the DevOps team: Murphy's law teaches a critical business request will come at the worst moment. So, for example, the request will come while you are firefighting a big problem, and so the team won't look at it for a long time. Once they notice the request, the DevOps team must start wasting time calling various other teams to guess who is really supposed to address that request, … a lot of other things will certainly happen so that everything that can cause problems and delays will take place until that request is delayed to the maximum possible degree. A lot of pain, stress, reputation, and money loss can be prevented by simply spending a couple of hours investigating the proper setup.

Long story short: don't be a blind executor; always carefully think about what you are doing and its impacts, even beyond the strict technical boundaries: we are not working in an ideal world – we are working for the business.

Co-workers are more than merely acquaintances

This is a golden rule: co-workers' personal problems affect how they do work and behave – you may pretend it's not like that, but this is the truth, since we are all humans. You must be empathic and encourage them to talk with you about the problems they want to share: this will just increase your mutual respect. In addition to that, you should spend some quality time with your co-workers – have lunch together, talk with them on topics that have nothing to do with work, … in the end if you are lucky, you may even make new friends. Working among friends is certainly better than working with acquaintances, and in my personal experience it happened to me sometimes to bring a friendship popped up at work also into my private life too (hanging out together, riding on motorbikes, and so on).

Summary

In this chapter, we explored one of the essential soft skills for a professional: being a good team player. The DevSecOps and DevOps disciplines emphasize collaboration between teams, which means being an effective team player.

Many issues in team dynamics can stem from poor communication and negative attitudes. This observation is supported by various studies and experiences in organizational behavior, and I can confirm it from my personal experience: I have found (and continue to find) that the root cause of most problems in team settings is a lack of communication, inadequate communication skills, and negative attitudes toward teamwork.

I hope this chapter has blown up your mind making you understand the importance of this critical soft skill, which is not only vital in DevSecOps and DevOps, but also essential for working effectively with others in any context.

By developing these skills, you will not only improve the quality of your work but also enhance your personal well-being, as the time you spend at work constitutes a significant part of your life.

CHAPTER 2

Solution Lifecycle

The DevOps discipline spans the entire solution and service lifecycle, often requiring DevOps and DevSecOps specialists to interact with development, security, QA, and operations teams. Understanding the correct terminology and foundational concepts is essential for effective collaboration and implementation of DevOps and DevSecOps practices.

In this chapter, we will explain the facilities you must leverage, such as the Configuration Management Database (CMDB), Definitive Media Library (DML), and Source Control Management (SCM). We will also discuss how the lifecycle workflow spans across various environments, including laboratory (Lab), development (Dev), testing (Test), pre-production (Pre-Prod), and production (Prod).

Additionally, we will cover concepts such as resource grouping by security tiers, the different types of networks, and the solution lifecycle, which includes design, transition, operation, continuous improvement, and, ultimately, the final dismissal.

Mastery of these principles ensures that security is not an afterthought but a fundamental aspect of the entire lifecycle, leading to the development of more resilient and high-quality products.

This skill set is invaluable, as it is a critical element of the holistic skill set expected of DevOps and DevSecOps professionals.

This chapter provides a brief, blended summary of concepts drawn from established frameworks, including ITIL (published by AXELOS), the PDCA (Plan-Do-Check-Act) cycle by W. Edwards Deming, and the DMAIC (define, measure, analyze, improve, control) methodology from Six Sigma.

It adapts these frameworks to DevOps and DevSecOps practices, incorporating original examples and real-world applications not found in the original frameworks.

This chapter is intended solely as an introductory overview to explain the existence and basic application of these frameworks in a DevOps and DevSecOps context. This is not a comprehensive or official guide, and it does not replace the original sources. For a deeper understanding, I strongly recommend consulting the primary materials directly:

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_2

- **For ITIL**: *ITIL Foundation, ITIL 4 Edition* by AXELOS (available from AXELOS or authorized publishers)
- **For the PDCA cycle**: *Out of the Crisis* by W. Edwards Deming
- **For DMAIC and Six Sigma**: *The Six Sigma Way* by Peter S. Pande, Robert P. Neuman, and Roland R. Cavanagh or official Six Sigma resources from organizations like ASQ (American Society for Quality)

Readers are encouraged to explore these works for accurate, detailed information.

Basic Concepts

Before anything else, it is necessary to clearly define a few basic concepts to avoid misunderstandings. Please spend some time reading Table 2-1 even if you are already acquainted with the terminology.

Table 2-1. *Basic concepts*

Step	Description
Configuration Management Database (CMDB)	A database containing all the information that is worth tracking about the IT infrastructure and services. Entries are called *Configuration Items (CI)*. A good CMDB application is also capable of inferring the dependencies across the various Configuration Items.
Definitive Media Library (DML)	It is a repository used to store and protect the corporate's authorized software versions, packages, and installation media. The best practice is to set it up to be `Write Once Read Many`, to always preserve software versions for reinstalling them as necessary.
Version Control System	An application specifically designed to provide facilities that enable code versioning. The most popular currently used VCS is certainly *Git*, but there were previous and now legacy VCSs such as Subversion (SVN) and the very old Version Control System (VCS).
Source Code Management (SCM)	It extends the basic feature set of a VCS providing additional features such as a web UI for easily managing pull requests and release management. Modern SCMs often provide CI features such as GitHub Actions.

(*continued*)

Table 2-1. (*continued*)

Step	Description
Environment	It is a way of segregating resources by usage purpose into a promotion flow. The typical list of environments is 1. **Lab (laboratory)**: Used to run disposable items suitable for running even destructive operations 2. **Dev (development)**: Used for the development of applications or automations and for running unit tests 3. **Test (QA testing)**: Used to run unit tests, integration tests, and functional tests 4. **Prod (production)**: Used to run customer-facing services It is worth mentioning that some organizations also use the Pre-Prod (often called staging) environment – this is a special environment where to deliver a service and run smoke tests including load tests and security tests: if the service instance passes these tests, it is then promoted to production. Mind that from the availability requirements perspective, they are all productive except the *Lab* environment – the only difference between the others is the impact of the service outages: outages in *Dev* and *Test* affect the corporate productivity but do not affect customers. Outages in *Prod* directly affect customers – this means more money loss compared to *Dev* and *Test* outages and, even worse, reputation loss. The reason behind this segregation is limiting the impact of out-of-control situations: despite being rare, they can happen in both *Dev* and *Test*, since people are constantly making changes and (non-destructive) experiments all the time; this segregation should spare other environments from problems spreading to them. Segregation means that, as a general rule, no traffic is allowed between the environments, although a few tracked exceptions mostly due to systems or device management or provisioning purposes can be granted.

(*continued*)

Table 2-1. (*continued*)

Step	Description
Security Tier	It is a way of segregating resources by security zones – a very common rule is that traffic between two different security tiers is strictly forbidden, even if they belong to the same environment. Access to each single security tier is often implemented using jump stations and VPNs.
Management Network	It is a network used only for management purposes – the typical implementation is using jump stations connected to systems of the same environment – in real life it is not uncommon to see jump stations and management stations connected to multiple management networks of different environments of the same security tier using different Network Interface Cards.
Provisioning Network	It is a network used only for provisioning purposes: it often contains software repositories as well as SCM services. In real life it is not uncommon to see management networks used also as provisioning networks.
Service Network	This is the network where service endpoints are bound. Although this is against the best practices, very often in real life, you will see service networks used also for provisioning and management purposes.
Promotion Flow	It is the delivery process across the list of environments of the same security tier. From the service perspective, the promotion flow starts from the Test environment going through Pre-Prod (if available) and then to production. From the development perspective, the flow starts in Dev, promoting to Test only the developed software (or automations) that passes the unit tests. After being published in the Test environment's software repositories, the software is delivered so as to be able to run all the necessary tests. After passing the required QA tests and after being approved for promotion, the software can be released to the production repositories, becoming available to customers for their on-premises installations, or delivered to service instances to be provided to the customers as paid services.

(*continued*)

Table 2-1. (*continued*)

Step	Description
Corporate-Wide Solution Design Document	It is the document specifying the corporate-accepted software, tools, development languages, standards, and best practices every project in the corporation must comply with. These standards include but are not limited to taxonomy, naming schemes, availability requirements by purpose, security requirements by purpose, monitoring requirements, performance metrics collecting requirements, datacenter automation requirements, and so on. Every solution-specific Solution Design Document must comply with this. If it is necessary to use something not specified in it for the first time, the specific solution is first discussed with the enterprise architect who must first validate it and add it to the corporate-wide Solution Design Document.

Solution Lifecycle

Despite DevOps and DevSecOps professionals being typically involved only in the implementation and operation of solutions – the actual design is typically made by architects – it can also happen that they are involved through the whole solution lifecycle. This can happen especially to the most experienced ones. For this reason, an understanding of at least the basics of solution lifecycle can really help a lot to grow as a professional.

The solution lifecycle spans several processes going from the solution design to its final dismission.

Note The processes described below only leverage the ITIL framework but are not fully compliant with it: they are instead blended with real-life on-field experiences. If you are interested in the pure ITIL framework, please refer to its official books.

The Service Design Process

This is obviously the initial process in the service or solution lifecycle: here, since we are focused on DevOps and DevSecOps, we are obviously intentionally skipping the purely business processes.

One of the most used solution design frameworks is the Deming Cycle, also known as PDCA.

The Plan-Do-Check-Act (PDCA) cycle provides a structured approach to problem-solving and process improvement.

The cycle consists of four key stages:

- **Plan**: Identify a problem or opportunity for improvement and develop a plan to address it. This stage includes setting objectives, figuring out the necessary resources, and outlining the steps needed to achieve the desired outcome.
- **Do**: Implement the plan on a small scale to test its effectiveness. This involves executing the planned actions, collecting the outcome data, and documenting the process.
- **Check**: Evaluate the outcomes of the "Do" phase by analyzing the data collected to determine whether the objectives were met and identifying any discrepancies between the actual outcomes and the expected ones.
- **Act**: Take action based on the findings from the "Check" phase. If the plan was successful, we can then standardize the process and implement it on a larger scale. Conversely, we may revise the plan and go through the cycle again.

Drilling down into more specific details, designing a solution requires going through the series of steps in Table 2-2.

Table 2-2. *Service design steps*

Step	Description
Gathering Requirements	This is the very first step: here you must interact with the *product owner* figuring out the project's requirements – mind that the product owner is always somebody from the business, so it's your responsibility to turn the business requirements into technical requirements. This is a very critical step – do not underestimate it, and especially if you are working *Agile*, remember to always define the project's scope, or sooner or later you won't be able to satisfy later requirements.
Challenging the Requirements	Sometimes the requestor's requirements hide misconceptions they may have on a specific topic: you must do your best to refine requirements to avoid wasting time and money to satisfy unneeded things or avoid doing them in the most efficient way.
Find the Right Products	Unless you have been explicitly asked to develop your own product (e.g., for selling it), avoid reinventing the wheel: always have a look to find out which products already available in the market more closely fit the requirements – sometimes you will even end up combining more products, developing the integrations and the missing parts as necessary.

(*continued*)

Table 2-2. (*continued*)

Step	Description
Initial Design	Draw up an initial design based on your guess from the available product documentation (mind not to definitively trust them), guessing all the necessary integration and custom software to be developed to meet the unsatisfied requirements. There are several aspects to be considered while designing, such as • **Maintainability**: The design will very likely be improved and changed during the project implementation and service lifetime to follow changes in business needs. Maintainability means that future changes must be cost effective, that they are not hard to implement, and that they must not require excessive reworks: this way the service is very likely to remain profitable and in line with the market's price. From the tech staff perspective, it is also an advantage, meaning less stress and frustration. • **Security**: A service (aka asset) is part of the value chain, so it MUST undergo a risk assessment and a risk treatment before going live. Taking this into account in the initial design can speed up passing that check avoiding reworks for remediations that may even severely affect the service maintainability degree. Pay particular attention to having your solution always compliant with the corporate-wide Solution Design Document – of course exceptions are possible, but they must be tracked and accepted by the enterprise architect and by the security team. When designing the infrastructure, always take care of requirements in terms of data integrity, confidentiality, service availability, and service capacity. It is also very important to assign straightforward, meaningful, and unique names to components, to be able to easily refer to them.

(*continued*)

Table 2-2. (*continued*)

Step	Description
Set Up a Lab Environment	It is always necessary to set up an environment with disposable instances and settings to play with for researching and testing the products and the developed integrations in a disruptive way. Mind the lab must never be decommissioned: it is used throughout the product or service lifecycle as a safe environment to run destructive or potentially destructive tests as necessary (e.g., simulating a risky upgrade). The lab is also a safe place to test all the delivery and configuration automations, such as Ansible roles and playbooks, Helm charts, and so on.
Live Demo	This is a critical step that can block or delay the project: you must show to the stakeholders a working demo with a meaningful subset of features that enable them to figure out that every requirement is achievable – the outcome can of course lead to reworks in the lab to meet needs that popped up during the demo – hopefully using the same products – and of course to other demos.
Draw Up the WBS Document	After a successful demo where you got the approval to go on, you need to create the Work Breakdown Structure (WBS) document that roughly put is the list of steps necessary for delivering the project to production.
Guess an Estimate	Once the WBS is completed, you must be able to document the estimated effort time, elapsed time, and costs for implementing the service.
Go or No-Go	This is much more a business and financial step – it is used to make an informed decision if it is worth the effort to go on with the project in total, if implementing only part of it, which one to do first, and so on.

Note When entering this process from the continuous improvement process, the "initial design" stage becomes "improving the current design."

During the service design process, there must be documents produced that will later be used as a reference during the implementation.

Table 2-3. *Service design documents*

Step	Description
Project Data Sheet (PDS)	This is a short document (often less than three pages) that must be produced at the very beginning of the project. It is a business document used for highlighting the project aim and overall scope, the business values it produces, the foreseen milestones, roughly estimated costs, and so on. Whilst this document is mostly business oriented, it must also provide the bare minimum level of technical details to have these business-oriented people be able to figure out the outcome. It is typically completed right before the Go or No-Go step.
High-Level Design Document	This document is used to describe the solution from the architectural perspective, identifying components and their dependencies, describing the required platforms and environment (operating systems, databases, and so on), as well as providing a high-level description of the backup and restore processes.
Solution Design Document	This is a more detailed document, mostly used to better describe the components listed in the High-Level Design Document. It describes the purpose and expected outcome of each single component in quite a summary way, whereas it provides very detailed information on the standard communication interfaces to be developed (endpoints, input formats, output formats, supported authentication and authorization mechanisms, and so on).
Work Breakdown Structure (WBS) Document	This document is mostly targeted at project management, highlighting every step necessary for delivering the project to production: the list must clearly identify for every single step the kind of professionals involved, the foreseen elapsed time and cost, and the steps that are prerequisites (providing dependency information is very important, since it enables working in parallel for achieving a quicker delivery). Although optional, it is of great help also providing a timeline without dates, showing the estimated progress of the project step by step, specifically highlighting which steps can work in parallel within each single timeframe.

Note When entering this process from the continuous improvement process, the above documents are of course amended to contain the details of the improvements.

The Service Transition Process

The next process is service transition: this is the phase where the set of service instances to be delivered across the environments is designed and deployed, delivering one environment at a time accordingly with the promotion flow.

Note When entering this process from the continuous improvement process, the current service design gets improved either in a new instance or in an online upgrade of the current instance, depending on the specific use case. This involves going through most of the steps below, this time focusing on the specific improvements.

The service transition process steps are summarized in Table 2-4.

Table 2-4. *Service transition steps*

Step	Description
Define SLA (or OLA)	Define the instance-specific Service Level Agreement (SLA) or, if the service is available within the organization only, define the instance-specific Organizational Level Agreement (OLA). These are used to clearly define the availability level, support team response time, and so on. The referenced agreement is not service specific: the best practice is to have a short list of corporate-wide SLA and OLA and apply the most relevant one.
Define the RACI Matrix	Identify the responsible team, the accountable team, the consulted persons, and the informed persons, as required by the ITIL framework. The RACI Matrix is typically the same across every environment.
Design the Delivered Instance	After having defined instance names consistent across the environments, using the High-Level Design Document as a reference, design the actual service instances to deliver. This task really depends on the available software you have: if your CMDB is smart enough to provide visual renderings and automatic dependencies, you may just enter the information in the CMDB. If you are not so lucky, you will also need to create a Solution Delivery Document.

(*continued*)

Table 2-4. (*continued*)

Step	Description
Capacity Check	Before moving to the next steps, you must also ensure they have enough capacity and budget to deliver and maintain each service instance – OK, I admit it: this is not a DevOps or DevSecOps responsibility, but it is always best to ensure the governance did this; otherwise some instances may not have enough hardware resources to properly run or, even worse, steal capacity from other services.
Work Organizational Structure (WOS)	Using the Work Breakdown Structure as a reference, draw up the Work Organizational Structure, clearly identifying people and teams responsible for each task listed in the WBS.
Design the Processes	This step is used to design the Feature Request (or Service Request) and submit a bug (or Incident Report) processes – if this step is skipped or poorly implemented, then users won't be able to easily interact with the service provider: this chaotic situation just leads to both customers/users and developers/operations/support teams being frustrated, delays when solving problems, and definitely reputation as well as money loss. When dealing with services consumed within your own corporation, if they are using a cost center, remember to include it as mandatory information in Service Requests to always be able to bill service usage to the correct entity. Another hint is to always ask the security team to review the designed processes, just to avoid them being blocked right before going live or even after they go live.
Develop the Automation Tools for Delivering the Service and Deliver It	During its lifecycle, the service will be delivered multiple times, updating it for applying patches, adding new features, or dismissing the no longer needed ones. Manual delivering is slow, not cost effective, and error-prone, so you must implement a delivery process that leverages automation tools such as Ansible or Helm. This step involves developing the necessary automations and documenting the configuration management facilities used to deliver the service, as well as the ones to be used to update it, suspend it, backup and restore it, and of course decommission it when it is no longer needed.

(*continued*)

Table 2-4. (*continued*)

Step	Description
Implement Monitoring Probes and Metrics Collecting	Monitoring probes are not necessary only during service operations – they are used also when rolling over updates to make sure the rolled over version is properly operating before opening it to live traffic. As for metrics collecting, they are not just the only reliable way to identify the performance problem's culprit at the application level, but they also enable automatic horizontal scaling by defining thresholds in orchestrating suites such as Kubernetes.
Training	It is ridiculous having to mention this, but before the handover it is mandatory to provide at least a basic training to the operational teams, showing them how to operate the service, how to use the support tools if any, and where they can find the official written documentation.
Publishing	This is the step where the service instance is delivered. When delivering to the production environment, this step is often called Go-Live – in this case, the service is advertised and becomes officially available and supported. Before Go-Live, it is very common to run some smoke tests too.

During the service transition process, it is necessary to produce a few documents that will be used as a reference during the delivery step.

Table 2-5. *Service transition documents*

Step	Description
Solution Delivery Document	This document, necessary only if the CMDB is not smart enough to provide visual renderings and automatic dependencies, contains the instance-specific information across the environments the service is delivered to. For example, the hosts running the service components, the database hosts where the databases are stored along with the database names, the involved web application firewalls and reverse proxies along with the related virtual hosts, and so on. It is useful not only during delivery, but also during incidents to have a quick understanding of the topology and components: this often speeds up troubleshooting since support teams do not have to waste time guessing topological and infrastructural information.
Work Organizational Structure	Based on the WBS, it clearly identifies people and teams responsible for implementing each task listed in the WBS. It must also contain a timeline with the delivery date foreseen and milestones.
Processes	This document describes each single process to raise a Feature Request (or Service Request) and submit a bug (or Incident Report) process. It is used by the guys who configure the ticketing systems to configure them in compliance with the described processes.

The Operating Process

The operating stage starts right after the service becomes available to the public for the first time: during this stage, according to the processes defined in the previous stage, there will be raised Service Requests, Feature Requests, and Incident Reports. The support and operations teams during this stage will write and maintain the Frequently Asked Questions (FAQ) used to help both the service users and even themselves to quickly find solutions to recurrent problems. The operations teams also write the best practices, deriving them from the workarounds and permanent fixes of Incident Reports and service outages.

This process continues until the service is decommissioned.

The Continuous Improvement Process

This process combines parts of both the service design and service transition processes to deliver improvements to the running service.

Improvements are based on

- Approved Feature Requests
- Specifically acknowledged submitted bugs
- Best practices

A framework often used in the continuous improvement process is DMAIC (define, measure, analyze, improve, control).

DMAIC is a data-driven, structured methodology that originates from Six Sigma, which emphasizes thorough data analysis and structured improvements, making it well suited for continuous improvement processes, including those in DevOps.

The framework is particularly effective in DevOps solution improvement processes, as it involves analyzing existing processes, identifying bottlenecks or issues, implementing iterative changes, and controlling those changes for sustainability. This structured approach aligns well with the needs of continuous integration/continuous deployment (CI/CD) pipelines and automation setups.

It consists of the following phases:

- **Define**: This initial phase focuses on clearly defining the problem or opportunity for improvement. Key activities include identifying project goals, understanding customer requirements, defining the project's scope, and creating necessary documentation (such as a project charter) to initiate the project.
- **Measure**: In this phase, the current performance of the process is measured. This involves collecting relevant data, identifying key performance indicators (KPIs), and establishing a baseline to compare against future improvements.
- **Analyze**: As the name suggests, the goal in this phase is to identify the root causes of the problem. This includes conducting data analysis, identifying patterns and trends, and performing root cause analysis using techniques like the 5 Whys or fishbone diagram.

- **Improve**: This is the actual improvement phase, where solutions to the identified problems are developed, tested, and implemented. It involves brainstorming potential solutions, piloting them, and rolling out successful changes across the process.
- **Control**: The final phase focuses on sustaining the improvements made. This includes monitoring performance through KPIs, standardizing processes to ensure consistency, and implementing control plans to prevent regression to old habits.

Summary

In this chapter, we explored the solution and service lifecycle, defining key concepts essential for mastery. We clarified terminology, discussed necessary facilities, examined the lifecycle workflow across various environments, and addressed resource grouping by security tiers, network types, and the solution lifecycle stages: design, transition, operation, continuous improvement, and final dismissal.

For DevOps and DevSecOps professionals, a solid understanding of the solution lifecycle and its terminology is crucial for effective communication and collaboration within teams. Familiarity with industry-specific terms allows professionals to articulate ideas clearly and engage meaningfully with stakeholders, developers, and operations teams. Grasping the lifecycle – from conception and development to deployment and maintenance – enables practitioners to anticipate challenges and implement best practices at each stage. Additionally, the ability to design robust solutions that meet both functional and non-functional requirements is vital for creating scalable, secure, and efficient systems. This comprehensive knowledge enhances problem-solving capabilities and fosters a proactive approach to innovation and continuous improvement, ultimately leading to successful project outcomes and satisfied customers.

CHAPTER 3

An Overview of Agile and Lean

In the IT world, several methodologies have been developed to facilitate development, enhance communication, coordinate service desk operations, and achieve overall success. Besides ITIL, which primarily focuses on managing IT services and improving support and service levels, the most prominent methodologies in the DevOps and DevSecOps fields are Agile and Lean.

Understanding these methodologies helps professionals navigate the complexities of software development while integrating security throughout the process. Therefore, it is crucial for every DevOps and DevSecOps professional to have a solid understanding of both.

A common question is which methodology to use, and the answer is: it depends. Both Agile and Lean aim to achieve business goals and delight customers with competitive products of the highest possible quality. However, they serve different purposes and tasks. While Agile is a subset of some Lean principles and practices, Lean encompasses a broader scope, as its approach is not limited to mitigating time loss; it addresses all types of waste, including money, labor, energy, and more.

We will begin our discussion with Agile, as it is the methodology you are most likely to encounter in the DevOps and DevSecOps fields.

This chapter provides a brief, blended summary of concepts drawn from established methodologies, including Agile (from the Agile Manifesto) and Lean (as adapted for software development). It is intended solely as an introductory overview to explain the existence and basic application of these methodologies in a DevOps and DevSecOps context. This is not a comprehensive or official guide, and it does not replace the original sources. For a deeper understanding, I strongly recommend consulting the primary materials directly:

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_3

- **For Agile**: "Manifesto for Agile Software Development" (freely available at agilemanifesto.org under a Creative Commons license)
- **For Lean**: *Lean Software Development: An Agile Toolkit* by Mary Poppendieck and Tom Poppendieck

Readers are encouraged to explore these works for accurate, detailed information.

Agile

It is a project management methodology specifically tailored and aimed at improving software development productivity. It was started in 2001 by publishing the Agile Manifesto (https://agilemanifesto.org/principles.html) containing the following 12 *principles*:

1. Our highest priority is to **satisfy the customer** through early and continuous delivery of valuable software.
2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. **Business people and developers must work together daily** throughout the project.
5. **Build projects around motivated individuals**. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.
7. **Working software is the primary measure of progress**.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to **maintain a constant pace indefinitely**.

9. Continuous **attention to technical excellence and good design** enhances agility.
10. Simplicity – the art of **maximizing the amount of work not done** – is essential.
11. The best architectures, requirements, and designs emerge from **self-organizing teams**.
12. At regular intervals, the **team reflects on how to become more effective**, then tunes and adjusts its behavior accordingly.

– Agile Manifesto

When doing Agile development, the product is delivered to the customer in small increments, each time gathering feedback that is synthesized and used to inform the next stages of the process.

This step-by-step approach provides several benefits: besides resulting in a quick time to market, it enables the development team to quickly react and adapt to changes depending on the feedback received. But most of all, it enables one to adapt and change the next deliverables at any step depending on feedback, market conditions, corporate obstacles, etc., and to supply only products with features relevant to the market.

There are several methodologies that can be applied to Agile principles, such as Scrum, extreme programming (XP), feature-driven development (FDD), etc. In this book, we quickly examine only Scrum, since it is the most used one.

Note Read carefully the second principle *Welcome changing requirements*: I often see people interpreting it as "do not spend too much time identifying the requirements and let's begin to develop instead: anyway, we have to welcome changes also in late development." I strongly disagree with this very dangerous approach, which conflicts with the ninth principle. We'll see how dangerous this attitude is shortly below.

Lean

Lean is a philosophy born in Japan in the mid-1950s at Toyota mainly aimed at loss reduction and sustainable production. In the 2000s, Lean was adapted for software development by Mary and Tom Poppendieck who related it with seven initial Lean principles and Agile philosophy.

Lean principles are a subset of system thinking that requires a never-ending cycle made of action loops that consist of learn, measure, build, and thoroughly test. In addition to that, Lean pays particular attention to customer feedback considering it the ultimate value to be used to continuously improve. It is a smart development that improves everything by eliminating anything that doesn't bring value to the customer. This approach, besides reducing the time to market, helps to figure out what the market wants and adjust the direction from time to time. The most used Lean framework is Kanban.

My Personal Feedback on Agile

This is my personal point of view after some years in the field. My personal experience with Agile projects governed by people interpreting the principles in a too strict way has been nearly always very disappointing: most of the solutions delivered were wobbly since the first release, becoming worse from the maintainability and availability perspective at every rollover. The more complex the project is, the more likely this problem is punctually raised.

I think the culprit is not really the framework itself, but rather

- A misunderstanding of the meaning of the second principle most people have – more details on this are provided later.
- Blindly believing that every person can really agree on and define common protocols and standards for the components they are working on only leveraging spoken communications. Besides the missing tracking record of each decision and interactions (read it as "no one is responsible for anything"), it leads to delays due to misunderstandings and misconceptions and unnecessary reworks delaying the delivery time. In addition to that, it increases the chances of having bugs.

- Having the Scrum master role often assigned to project managers without enough skills on the specific project's technologies and often in the specific technology field in general, demanding them to work as architects. This nearly always leads to issues such as missing or underestimating problems, choosing wrong or unsuitable products, and defining too strict milestones – causing excessive stress and frustration in team members. A typical way to cut corners I often saw being used while struggling to meet underestimated milestones is to procrastinate the implementation of exception checking code and bug fixing: the sad truth is that they are accumulating a huge amount of technical debts, until they reach the point of not being able to address them. Then the only way out is delivering a solution that runs, … but does not really work properly, completely relying on operations teams' efforts to quickly fix service outages and runtime errors – as if this was cost-free (this is just hiding dust under the carpet, not to say cheating).
- Thinking that Solution Design Documents are not necessary, having people just implement things the way they think it is best, … this is another great way leading straight to technical debts: requisites in user stories must always be challenged and optimized, … every developer will do this challenging in their own way, depending on their skill and experiences, so someone will do a great job, someone will do so and so, someone won't do this at all, … and someone will just make a mess: the only guaranteed outcome is that instead of having a clean and tidy design, you will get a chaotic mixture of styles.

For this reason, before digging into the Scrum framework, I'm giving you the below personal warnings.

Define a Scope

As soon as you get involved in a project, you must ask the product owner to define the macro scope of the product; this is the most important requirement: neglecting it always leads to catastrophic outcomes sooner or later, because of the blind application of the second principle.

The macro scope is necessary to define the boundaries to avoid later requirements in contrast to the pillars of the overall design: when dealing with requests that would involve reworks and redesigns that risk to compromise the product maintainability and reliability, referring to the macro scope is the only way out to make businesspeople admit they are asking something that cannot be fulfilled in this project. Not defining it means, because of the second principle, giving them the right to ask for whatever change.

When a request is out of the macro scope, … it just means it is the time to start another Agile project.

An Agile Project Without a Macro Scope

To see how bad things can go, as an example, consider a customer just requesting an Agile development for a *means of transport*. The deliverables iteration would be nearly like this:

1. **Deliver a bike**: The feedback is to improve it into something that can go faster and that can move by its own – you add a motor, and you rework it as a motorbike.
2. **Deliver a motorbike**: The feedback is to improve it into something that can spare people from getting wet if it rains and to make it suitable for up to five people – you transform it into a car.
3. **Deliver a car**: The feedback is to improve it into something suitable to transport 20 to 50 people – obviously you turn it into a bus.
4. **Deliver a bus**: The aim is to improve it into something capable of transporting at least a few hundred people. Of course it's time to improve it into a train. This time the change is challenging. You do not rework only the means of transport; you also need to deliver a suitable infrastructure for running the train, … a lot of work, but still doable.
5. **Deliver the train**: The feedback is to improve it, making it fly, … and therein lies the rub! For the second principle, you must satisfy the request; otherwise … you are not Agile.

The technical logic suggests that it's not worth the effort, that it's unlikely to work, and even if it works it's risky and a lot of other very good reasons which your customer won't be able to understand and agree with for a very simple reason: they are from business, so they are not required to understand anything about technology.

So, it's all your fault: you fail to meet their expectations – you cannot improve the product in the next cycle. So, you are guilty of not having defined the macro scope: if you did it at the very beginning, such as agreeing to a project for developing land vehicles, you'd be able to always satisfy requests.

In this example you are lucky, since the customer waited until the fifth iteration before asking to make the vehicle fly, … just think if they asked to make the car or the bus fly. Of course you may probably do that, but it is very likely not worth the effort because of the rework; it would be risky for customers using that product and the market will probably just prefer products specifically designed to fly since they are more cost effective and certainly work best in general.

If you agreed on the scope, you could just remind the customer that the project is for land vehicles: if they want something that flies, no problem – it's just a matter of starting a new Agile project, employing professionals with specific skills.

An Agile Project with a Macro Scope

Let's try the "agree on a macro scope" approach: after talking with the product owner, you agree on a project for delivering *flying vehicles*. This time the delivery iterations may be as follows:

1. **Deliver a hang gliding**: The feedback is to improve it into something with a cockpit.
2. **Deliver a glider**: The feedback is to improve it into something that can take off by itself.
3. **Deliver a single-engine airplane**: The feedback is to improve it into something bigger suitable to transport hundreds of people or heavy loads.
4. **Deliver a multiple-engine airplane**: The feedback is to improve it into something usable to refill the International Space Station.

Now your customer says that they need to use it to refill the International Space Station, … can you hear the buzzer? You are out again!

Kidding aside, I just wanted to show you that you must be very careful even when defining the macro scope: in this case, after understanding that the customer wanted something able to fly, the next question would have been "flying in the air or in space?"

Project Managers Are Not Solution Architects

In my experience too many times I saw project managers happily creating epics, stories, and tasks in project management software directly deriving them from the list of requisites, without dedicating enough time to do a proper analysis and drawing up a Solution Design Document – they call it *bottom-up approach*. It is the same as you ask a construction company to build your house, and they ask you to write your requirements, and then you see a guy having nothing to deal with construction engineering writing them on yellow stickers, giving them to masons, and they "implement" the parts they have been assigned designing them in their own mind. Can you figure out what a house that would be? And even then, their chances of success are higher than in the IT field, since conversely from the modern IT field, no one is also demanding them to be full stack (be mason, tiler, plumber, electrician, and furniture maker at the same time).

Be careful if you find yourself working on projects managed this way, because unless they are very small and trivial projects, the risk of failure or of building something ridiculously fragile is extremely high.

Summary

In this chapter, we provided an overview of Agile and Lean, the leading methodologies in the DevOps and DevSecOps fields. To complement theoretical concepts, I shared personal experiences with Agile's caveats and pitfalls, hoping to help you avoid them.

For DevOps and DevSecOps professionals, expertise in Agile and Lean is crucial for fostering a culture of continuous improvement and delivering high-quality software efficiently. Agile promotes flexibility and responsiveness to change, allowing teams to adapt quickly to evolving requirements and customer feedback. Lean principles focus on maximizing value while minimizing waste, streamlining processes, and optimizing resource utilization.

By integrating Agile and Lean practices, professionals can enhance productivity, reduce cycle times, and improve project outcomes. Mastery of these methodologies empowers practitioners to create efficient workflows, drive innovation, and consistently deliver value to customers.

CHAPTER 4

An Overview of Scrum and Kanban

As we discussed in the previous chapter, the most widely used project management methodologies in DevOps and DevSecOps are Agile and Lean.

In this chapter, we delve into the frameworks they leverage: Scrum for Agile and Kanban for Lean.

Scrum and Kanban frameworks are crucial for professionals because they foster a collaborative and efficient work environment that enhances team productivity and responsiveness.

Mastery of these methodologies enables professionals to effectively participate in Agile- and Kanban-managed teams, facilitating better communication, iterative development, and continuous improvement.

These soft skills are essential for integrating development and operations practices, ensuring that teams can adapt quickly to changing requirements and deliver high-quality software in a secure and timely manner, which is a core principle of the DevOps discipline.

This chapter provides a brief, blended summary of concepts drawn from established methodologies, including Scrum (from the Scrum Guide) and Kanban (as described in related frameworks).

It adapts these frameworks for DevOps and DevSecOps practices with original insights and real-world applications.

This chapter is intended solely as an introductory overview to explain the existence and basic application of these methodologies in a DevOps and DevSecOps context. This is not a comprehensive or official guide, and it does not replace the original sources. For a deeper understanding, I strongly recommend consulting the primary materials directly:

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_4

- **For Scrum**: *The Scrum Guide* by Ken Schwaber and Jeff Sutherland (freely available at scrum.org under a Creative Commons license)
- **For Kanban**: *Kanban: Successful Evolutionary Change for Your Technology Business* by David J. Anderson or *Kanban from the Inside* by Mike Burrows

Readers are encouraged to explore these works for accurate, detailed information.

Scrum

Scrum is an iterative incremental time-boxed methodology suitable to implement Agile, aimed to manage things to have them delivered at the right time.

Everything starts from a project with

- A clear (although not detailed) vision of what should be achieved, stated on business needs rather than in technical terms.
- Vague details on how and when to achieve the goal: they become clearer and clearer as the project moves forward.

Scrum Roles

There are three roles in Scrum as shown in Table 4-1.

Table 4-1. *Scrum roles*

Role	Description
Product Owner	He represents the stakeholders, and he is responsible for the product (or the service availability of that product) and for the return on investment (ROI). Because of these responsibilities he's the one granted to prioritize the product backlog using business value points to ensure that the most valuable functionalities are developed first. Effective product owners also seek input and feedback from customers, from designers, and from the team itself to optimize product delivery.
Team	They are the ones that turn concepts into reality: they guess how to turn the product backlog into an increment of functionality within each timeframe of the sprint (everyone in the team is jointly responsible for the success of the iteration) and they do implement things.
Scrum Master	He is not part of the team: he is responsible for the Scrum process, making sure everyone follows the rules, removing impediments for the team if necessary. Of course, project managers can be Scrum masters, but they must have the appropriate technical skills required by the project; otherwise you run into the risks I already mentioned.

Note Although a project manager can also be the product owner, this does not mean that these two roles do the same things: the project manager ensures that the project is delivered on time and that requirements are met. The product owner is more bound to the "soul" of the product: they're the ones that discuss with the stakeholders, explain the requirements, and accept or reject contents from the team. In other words, they're the ones that have the vision of the product.

Scrum Project

When dealing with a project, the very first thing to do is

- Gathering the requirements to get to what the project wants to achieve
- Defining a roadmap to implement it

Right after this, a prioritized list of deliverables, derived from the roadmap and from the requirements, is stored into the so-called *product backlog*: the most urgent items are put by the product owner at the top of the product backlog so that the team delivers them first. The overall work inside a project is split in smaller units:

- **Epics**: These are the most high-level work units – they are groups of smaller working units called stories. For example, the effort to be done to implement a set of features can be considered an epic.
- **Stories**: These describe a need – for example, a feature of the product. A story should describe the need from the user's perspective avoiding technicisms: it should always mention the user role that is needing the request to be fulfilled, the goal to be achieved, and why the user needs it (i.e., received benefit).

It is up to the product owner to decide which ones to complete first: maybe an entire epic, maybe some stories of different epics because they are needed by the business before the others. Uncompleted stories are put into and stay in the backlog: their state can be either

- New (freshly inserted)
- In progress
- Blocked

and so on; the key point you must get here is that there cannot be completed stories in the backlog.

Scrum Sprints

Scrum iterations are called sprints: these are the forecasts to complete user stories or other work units during a fixed amount of time. A sprint usually lasts one, two, or even four weeks (although two weeks is the most common duration): its lasting is highly related to the type of work – if we are talking of development and the work is split into big projects, sprints can be long. If we are talking of shorter goals, like system engineering tasks such as installing a new service infrastructure, sprints can be very short (usually one week is enough).

Note Consider that a sprint should be long enough to get something accomplished, but not so long that the team isn't getting regular feedback.

Scrum Ceremonies

Scrum requires some ceremonies: attendees are always at least the Scrum master, the product owner, and of course the team.

Note Do not think that ceremonies magically make things Agile – they only promote communications: this is only a piece of Agile principles.

Table 4-2. *Scrum ceremonies*

Ceremony	Description
Sprint Planning Meeting	Choosing the right work items for a sprint is a collaborative effort between the product owner, Scrum master, and team: before starting the sprint, the Scrum master should plan a meeting (about one hour for each week the sprint lasts for – e.g., a two-week sprint can have a two-hour sprint planning meeting). In the first part of the meeting, the product owner discusses the objective that the sprint should achieve and the items of the product backlog necessary to be completed to achieve that goal. The product owner makes regular reviews of the backlog before the sprint planning meeting to ensure its health: this is called backlog grooming, which is the Agile term for backlog refinement – this is not a "ceremony": however, this is an important ongoing process in which the product owner collaborates with the team that can of course lead also to scheduling further meetings if necessary. The aim is making sure that everything is going smoothly, for example, providing more details that clarify the implementation, splitting larger tasks into smaller ones, prioritizing or re-prioritizing tasks, identifying obstacles and risks, and figuring out how to sort them out. The team then • Sketches up tasks from the high-priority items and estimates the time to accomplish them • Commits to completing a certain number of stories by the end of the sprint: these stories and the plan for completing them by the end of the sprint is what is known as the sprint backlog It is wise to be careful pulling in only tasks and stories the team can complete by the end of the sprint, avoiding overestimating velocity: not caring about that just leads to a very likely failure of the goal. Even worse, the reaction is that after a few sprints people will overestimate time to be always sure to meet the goal. In addition to that, it is necessary to always consider risks: it is best to scatter risky tasks and stories across different sprints, to avoid having some of the sprints with too many risky things. And most of all, it is important to listen to the concerns the team may raise.

(*continued*)

Table 4-2. (*continued*)

Ceremony	Description
Daily Stand-Up	A daily recurrent, very short meeting (should last no more than a few minutes), typically in the morning – it is called stand-up because attendees should stand up … this helps keep the meeting short. Its purpose is to quickly inform everyone of what's going on across the team. It is not intended to provide detailed information, but rather to surface any blockers and challenges that would impact the team's ability to deliver the sprint goal.
Iteration Review	A 30- to 60-minute meeting at the end of the sprint the team uses to celebrate their accomplishments and to demonstrate the work finished within the iteration. Attention must be paid so that the demo is fully functional; otherwise its milestone is not achieved. The purpose is to get immediate feedback from project stakeholders (stakeholders are encouraged to attend this meeting). In addition to that, it is wise to give a celebratory feel, to have team members be proud of what they did and the achieved outcome.
Retrospective	A one-hour meeting right after the iteration meeting: it is used to provide quick feedback to improve the product and its development. Retrospectives help the team understand what worked well and what didn't.

Kanban

Kanban is considered as one of the best Lean methodologies in the software and IT industry. It has three basic items:

- Boards
- Lanes (lists)
- Cards

Note Every project or workflow must have its own Kanban board. As for the implementation to use, there is software like Taiga (or the more popular Atlassian's JIRA) which implements software Kanban boards, but even a simple blackboard is enough.

Kanban requires dividing the board into several lanes (some people call them lists): the simplest set is

- To do
- In progress
- Done

This is the simplest approach, but it is possible to add as many lanes as it is required to get a better governance of your project (of course, lanes are very specific to what your projects apply to). Examples of additional lanes can be

- Paused
- Review

and so on.

The last item is the card: it is the working unit to be tracked using Kanban.

Cards are initially put as to do and then will be moved forward throughout the workflow until they reach the end.

Note Remember that cards are working units: too big working units must be avoided since they will stay in progress for a long time. This can cause several problems: for example, other working units depending on the stuck one will get blocked until it is completed. In addition to that, this adds stress to the team speeding up a massive task – in general we can say that they add wait time, that is exactly what should be avoided. The rule of thumb is to assign a maximum amount of time the cards can stay as in progress, then cards must be sized so that they can fit in that amount of time: the team will work better, avoiding wait times, and everyone will get more confident since targets will be met very often.

Kanban Roles

Just as Scrum has the role of Scrum master, Kanban also has some specific roles for Kanban management as shown in Table 4-3.

Table 4-3. *Kanban roles*

Role	Description
Service Request Manager (SRM)	Its goal is bridging the gap between the business and the customer. He asks the individuals to improve the efficiency and effectiveness of the workflow. His main responsibilities are • Create an environment that brings value to the customers and to the business: to achieve this he gets feedback from the team, stakeholders, and customers so to prioritize work, taking in account of cost of delay, complexity, analyzed approach, technical risk market competency, and many more • Define and maintain process policies: to promote the underlying business value he introduces and manages CoS (classes of service) that rank work items according to priority. He establishes also sequencing policies (according to priority level of different CoS) that define how each CoS should interact with each other • Make sure that both the CoS and their sequencing policies are in place
Service Delivery Manager	They oversee the quality-of-service delivery, ensuring an appropriate response to the demands of the customer. They help the team work the best way by focusing on shortening the response to the market demand, on optimizing the delivery speed while reducing costs. Their responsibilities are as follows: • Ensure that the best Kanban policies are followed by the team members. • Check the Kanban board to make sure that no work item is pending or has been blocked for too long – if a certain task is being delayed more than expected, they check if there's a problem and also address the topic with the requester. • Promote continuous improvement with the workflow process: to achieve this they should be involved in meetings and activities such as the Service Delivery Review. • Identify loopholes of work items on the Kanban board and their root causes so that errors are not repeated more than once.

Kanban Meetings

Of course, there cannot be true governance without meetings –Kanban also requires them: they are specific to each board and of course a moderator is needed. Some of them are oriented to the service delivery itself, others are specific to improvements.

Table 4-4. *Kanban meetings*

Meeting	Description
Delivery Planning Meeting	This kind of meeting is scheduled at each delivery cadence and is aimed at communicating to entities receiving and accepting the delivery: within Kanban, when tasks are completed, often other dependent tasks become deliverable (maybe to other teams). The outcome should be to ensure a smooth transfer of work in progress and plan and decide which items to deliver. Be wary that this meeting is not universally defined in all Kanban implementations. Some teams may not have a formal "Delivery Planning Meeting," as Kanban emphasizes continuous flow and delivery rather than fixed planning intervals.
Daily Meeting	Somebody calls this meeting "walking the board": it's a very short (it should last no more than 15 minutes) daily meeting mostly aimed at identifying new events that risk being team blockers, but also to present what's going on. It's also the right moment to check if work in progress limits have been met, before pulling new tasks from the backlog.
Replenishment and Commitment Meeting	It's aimed at taking informed decisions (shared at the Daily Meeting), supported by the team and stakeholders. These decisions help to prioritize incoming work and keep a steady flow of work. Besides the team, it also includes the product owner, the product development management, and anyone who can provide help in scheduling, sequencing, or grouping tasks. It usually lasts 20-30 minutes and it can be seen somehow the same way as the Sprint Planning meeting.

(*continued*)

Table 4-4. (*continued*)

Meeting	Description
Strategy Review	During this (usually four hours) quarterly meeting, the overall business strategy gets reviewed and adjusted according to market and customer changes. It's here that potential large-scale problems should be identified along with their suitable solutions. Information and decisions for this meeting are suitable for finding key performance indicators (KPI). From the team perspective, typically this meeting is attended by representatives of customer-facing teams, but the actual attendees can vary by organization: for example, some organizations may include other stakeholders as well.
Service Delivery Review	This (usually 30-minute) bi-weekly improvement meeting is aimed at checking that the delivered service met the customer expectations. From the team perspective, only representatives of the teams attend this meeting, whose decisions and findings are then reported at Operations Review. Moreover, findings from this meeting can lead to actionable improvements in service delivery, which is a key goal.
Operations Review	The aim of this (usually two hours) monthly meeting is to ensure that work goes smoothly across the different teams: demand and capacity of each Kanban team are checked, also relying on findings of Strategy Review and Service Delivery Review. Decisions are then communicated at the Service Delivery Review, Strategy Review, and Risk Review meetings. Generally, no team members attend this meeting, but there can be exceptions, for example, to cope with operational challenges and capacity issues.
Risk Review	This (usually one to two hours) monthly meeting is aimed at identifying risks and bottlenecks before they substantially impact the workflow and to take the right steps to mitigate those risks. It considers the issues identified at the Operations Review and Service Delivery Review as well as the input from Delivery Planning Meetings. Everyone familiar with the recent blockers, even team members, attends this meeting.

Summary

In this chapter, we explored the Agile and Lean methodologies by examining the Scrum and Kanban frameworks they employ.

For DevOps and DevSecOps professionals, proficiency in Scrum and Kanban is essential for enhancing team collaboration and improving project management efficiency. Scrum's adaptability is crucial in fast-paced environments where requirements can change rapidly, while Kanban focuses on visualizing work in progress and optimizing workflow, helping teams identify bottlenecks and boost productivity.

Mastering these methodologies fosters a culture of continuous improvement, enhances communication among team members, and ensures projects align with business goals. Ultimately, expertise in Scrum and Kanban equips DevOps and DevSecOps practitioners with the tools to manage workflows effectively, respond swiftly to challenges, and deliver high-quality software solutions.

PART II

Operating with the System

Operating directly with the operating system constitutes the foundational layer of control in DevOps and DevSecOps engineering. Beyond the abstractions of cloud dashboards, web interfaces, and high-level orchestration planes, all execution ultimately resolves within raw operating system kernels. Without the ability to directly manipulate processes, navigate file systems, or orchestrate low-level network states, engineers lack the visibility required when production workloads halt or security breaches occur. Modern paradigms like GitOps and CI/CD pipelines are not self-sustaining abstractions; they are automated sequences executing systemic commands, typically within a system shell. Mastering this section moves past manual, interface-dependent configurations to establish absolute control over the execution environment. Rather than a regression into legacy administration, this section delivers the core engineering skills required to stabilize infrastructure, optimize runtime performance, and automate analytical defenses directly within the core execution layer.

CHAPTER 5

The BASH and Command-Line Utilities

The very first DevOps and DevSecOps technical pillar to acquire are the skills on the system shell and the commonly used command-line utilities: daily activities indeed involve writing shell statements, running shell scripts, as well as writing new ones and maintaining the current ones. This obviously requires knowing the most used command-line tools so to be able to use and combine them in command pipelines, saving them into batch files as necessary, or writing shell scripts to address more complex tasks.

Mastery of shell scripting represents a foundational step in the development aspect of the DevOps and DevSecOps landscape, empowering professionals to create robust automation solutions that improve deployment processes, system monitoring, and configuration management, ultimately contributing to a more agile and secure software development lifecycle.

Note As mentioned in the Introduction, the required baseline skill for reading this book is already having a basic understanding of Linux: this of course also includes very basic shell concepts that in this chapter we are taking for granted, to avoid wasting precious space on trivial topics, and focusing instead on the interesting and less straightforward ones.

In this chapter, after introducing the concept of shell and the most used command-line utilities, we delve into the Bourne Again SHell (BASH) from both the operational and scripting perspective, ending the chapter with a quite comprehensive BASH cheat sheet.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_5

Note In this chapter we start practicing with Linux on the lab machine. Since several applications used in the labs are published in the EPEL repository, we **must** enable it by running: `sudo dnf install -y epel-release`. In addition to that, to keep everything properly grouped, we use the `~/devsecops-pillars` directory as a workspace throughout the whole book (`~` is a shorthand for the user's home directory). Each chapter has its own workspace: for this chapter we use `~/devsecops-pillars/ch05` directory tree. From this point onward we refer to this directory as to the **chapter's workspace directory**. Create it now by running: `mkdir -p ~/devsecops-pillars/ch05`.

What Is a Shell

A shell is a command-line interface that allows interactions with the operating system by executing statements, running programs, and managing files. It acts as an intermediary between the user and the kernel of the operating system, interpreting user commands and translating them into actions that the system can perform.

The shells provide a small set of built-in functions, supporting the definition of variables and arrays as well as unnamed pipes and input and output redirection: this enables writing command pipelines to address even complex tasks by mixing built-in functions and applications, piping or redirecting the input and output of each of them as necessary.

To support automations, it is even possible to store a list of pipelines into batch files to run all of them at once by running the batch file.

In this book, we focus on the Bourne Again SHell (BASH).

Note This chapter contains a lot of statements and command snippets: to avoid messing up your lab system, change to the chapter workspace directory by running `cd ~/devsecops-pillars/ch05`.

Commonly Used Command-Line Utilities

Before delving into the BASH, in this section we have an overview of the most frequently used command-line utilities.

Table 5-1. *Commonly used command-line utilities*

Utility	Description
man	Display the manual page with the documentation of the specified application. For example, to view the manual page of the `ps` command, type `man ps`.
ps	List the running processes. To list every running process, type `ps -aux`. You can easily filter the process list and customize the output as necessary. For example: `ps -C sshd -o uname,pid,command`.
date	Print the system date and time – mind that you can format the output as necessary – for example, `date '+%Y-%m-%d %H:%M:%S'`
echo	Prints to the standard output the specified message. A commonly used option is `-n`: when this option is used, the message is printed without the trailing newline special character.
read	Reads data from the standard input and stores it into a variable. A commonly used option is the `-s`, which disables echoing the typing – for example: `read -p 'Password: ' -s PASSWORD` reads the typed content and stores it in the PASSWORD variable, preventing the typed content from being printed to the screen. If used with the `-a` option and a here-string (`<<<`), it can read the contents of a string and store them into an array.
cat	Concatenates and displays the contents of files. For example: `cat server.crt server.key > server.pem` concatenates the contents of the `server.crt` and `server.key` files: the content is then redirected using the `>` **shell redirect operator** to the `server.pem` file, creating it if it does not exist or overwriting its contents if it already exists. If necessary, the `cat` command-line utility can also receive input by piping or using `heredoc` (`<< EOF`). Most of the time the output coming from `cat` is then redirected to a file or piped to other command-line utilities to implement complex pipelines.

(*continued*)

Utility	Description
touch	Touch a file, updating its access and modification times. If the file does not exist, it creates it as a new empty file. If necessary, it can update only the access time (`-a`) or modification time (`-m`) and even copy the time from another file or set any arbitrary time.
truncate	Shrink or extend a file to the size specified by the `-s` parameter. A very common use case is emptying log files using the statement `truncate -s 0` followed by the path to the file to truncate. Remember to always kill with the proper signal number the process writing the truncated log, to have it hang up and reattach its file descriptors. The signal to use typically is SIGHUP or SIGTERM, but this varies, so check the specific application's documentation. Forgetting this leads to a discrepancy between the space consumed on the mounted filesystem and the size reported by tools such as *du*.
head	Prints the first ten lines of the input – you can alter the number of printed lines by passing the `-n` parameter followed by the number of lines.
tail	Prints the last ten lines of the input – you can alter the number of printed lines by passing the `-n` parameter followed by the number of lines. If the number is prepended by a + sign, it prints the full file skipping the first lines. Another commonly used option is `-f`, used to follow the file printing the new contents as soon as they are appended.
df	Report the amount of disk space free and used on the specified filesystems – if no filesystem is specified, it reports the values for every single filesystem. The `-h` option provides human-friendly output, but it is possible to explicitly specify the unit to use, such as `-k` for Kilobytes.
du	Estimates and reports the disk space used by the specified files or directories – commonly used options are `-s` to print the cumulative space of all objects (including children when dealing with directories). The `-h` option provides human-friendly output, but it is possible to explicitly specify the unit to use, such as `-k` for kilobytes. **Note** Mind that when using `du` to estimate and report an entire filesystem – such as `du -sk /var` – the guessed value has a discrepancy with the used value reported by `df` (`df -k /var` in this example). That is expected, since `du` counts only the actual used space of the objects the user has access to, so metadata, deleted files still in use, and filesystem reserved space are not included in the count.

(*continued*)

Utility	Description
watch	Runs a specified command at regular intervals and displays the output in the terminal. It is possible to specify the interval in seconds between the runs using the `-n` option. For example, `watch -n 5 'df -h /var/lib'`. It is even possible to specify also the `-d` option, to have it highlighting the differences with the previous run.
wc	Count the number of words, characters, or lines in the provided input – commonly used options are `-m` to count the characters, `-l` to count the lines, and `-w` to count the words.
sort	Sorts the provided input – the most used options are `-r` to reverse the sort order, `-k` followed by the number of the key (the column) you want to use while sorting, and `-n` if you want to sort numbers accordingly with their numeric value.
cut	Cut the input extracting a specific field – the most used options are `-f` to specify the number of the field to extract and `-d` followed by the character you want to use as the delimiter.
tr	Transforms the specified character into another one in the given input. If used with the `-d` option, the specified character gets deleted. It also supports special tokens to specify spaces `[:space:]`, uppercase `[:upper:]` or lowercase `[:lower:]` characters. For example: `tr [:space:] "\t" <<< "space separated text"` replaces with a tab character every space in the message.
find	Find files or directories matching specific criteria, such as the kind (file, directory, symlink), having a specific name or matching a pattern. Valid search criteria are also the last accessed or changed date and so on. Be wary of the `-exec` option, which can be easily exploited to run statements on each single found file, and the `-delete` option, which deletes the found file (useful, e.g., while housekeeping).
ssh	SSH is used to securely connect to remote hosts, allowing to log in, execute commands, and transfer data over encrypted channels. Upon successful login, SSH can spin up any application on the remote host (by default it launches the user's default shell). Additionally, it enables piping streams by running statements on the remote host receiving data streams from statements run on the local host, facilitating the direct transfer of data in a secure manner. On some distributions it is linked to the PAM library: in this case additional features are available, such as time-based access enforcing. For example: `ssh remote.host.fqdn`.

(*continued*)

Utility	Description
scp	Securely copy using the SSH protocol files and directory trees to and from remote hosts. For example: `scp ~/foo.txt remote.host.fqdn:~/`.
rsync	Efficiently and securely transfer and synchronize files and directories between local and remote systems using a delta-transfer algorithm to minimize data transfer: this enables copying only the differences between source and destination, making it fast and bandwidth efficient. For example: `rsync -av --delete /var/www/ other.host.fqdn:/var/www/` synchronizes the contents from the `/var/www` directory of the current host to `/var/www` on the remote host. This includes also deleting files on the remote host no longer available on the source host.
mail	A twofold command-line utility that can be used both to send email and to read email from the user's own mail spool directory. Please note that through the time the original `mail` application evolved into `mailx` and `s-nail`, primarily to provide its capabilities to directly connect to external SMTP servers without leveraging on a locally installed and running MTA such as `sendmail`.
curl	cURL stands for "Client for URL": it is a command-line utility that can be used not only for downloading content, such as files, but also for interacting with HTTP-based APIs.
gzip	Compress and restore a file using the DEFLATE compression algorithm.
bzip	Compress and restore a file using the Burrows-Wheeler transform and Huffman coding – it provides a better compression than `gzip`, but with slower performance.
xz	Compress and restore a file using LZMA (Lempel-Ziv-Markov chain algorithm) or LZMA2 algorithm – it provides a high compression ratio.
tar	Manages the archiving (creation of the archive, adding, removing, and updating contents) of files and directory trees into a single file container – often called tarball. The tarball can be also compressed using any of the above compressions (the options are `-z` for gzip, `-b` for bzip, `-J` for xz). Be wary that, despite *tar*, by default, preserving the POSIX access permissions of files and directories, it does not save the POSIX ACLs. If you need them too, you must pass the `--acls` option both when archiving and when extracting.

Their usage is so trivial that it is not worth the effort to explain them one by one in detail: if necessary, you can experiment yourself with their syntax – there are also plenty of examples on the web. In addition to that, this book contains many examples with pipelines made by combining these utilities.

Note As we said, combining these utilities is the cornerstone for shell scripting: conversely from development languages, shells do not have libraries, so tasks are achieved by combining external command-line utilities into pipelines.

To demonstrate why understanding these utilities is crucial for creating powerful command pipelines, let's examine this example:

```
find /opt -type f -mtime +30 -exec du -sk {} \; | cut \
-f1 | awk '{total=total+$1}END{print total/1024}'
```

This pipeline finds beneath the `/opt` directory tree the files that have not been changed in the last 30 days and computes their overall size using *awk* – you can of course replace the `/opt` directory with anything you want to use as the entry point for the search.

Note Don't mind if you don't understand this `awk` statement: we will thoroughly discuss `awk` in Chapter 10.

You can check the outcome, and if the consumed space is beyond the acceptable, you can easily get rid of the found files older than 30 days by adding the `-delete` option – this will delete every found file, so use the `-delete` option with caution.

When running a find command with the `-delete` option, it's important to remember that `find` follows symbolic links by default. This means that if a sym-link points to a target outside the specified path, find may inadvertently delete contents in other locations, including files on different mount points.

To better control the path followed by find, you can use options such as `-P`, `-xdev`, and `-maxdepth`:

- `-P`: Prevents `find` from following symbolic links
- `-xdev`: Restricts `find` to the current filesystem, preventing it from descending into directories on different mount points
- `-maxdepth`: Limits the search to a specified number of directory levels, which can help avoid unintended deletions in deeper subdirectories

Shell Concepts

In this section, we explain the fundamental concepts necessary to operate with a shell and to write effective scripts.

Commands and Pipelines

Note Every content in this book is supposed to be run on a disposable and isolated system with only access to the Internet. If you have not read it yet, read the *Introduction* section at the beginning of the book because it contains important information you must read before proceeding with the rest of the book.

The shell, often referred to as a command-line interpreter, serves as an interface that awaits user input for commands, which can be either built-in functions or installed applications.

For instance, the command

```
ls -a1 ~
```

lists the contents of the user's home directory. The shell allows users to tackle complex tasks that cannot be accomplished by a single command by combining multiple commands into a pipeline. This involves piping the output of one command to the input of the next using the pipe | character as a connector.

For example, the statement

```
ls -a1 ~ | tr [:lower:] [:upper:]
```

not only lists the contents of the user's home directory but also transforms that output to uppercase.

In this case, the first command (`ls -a1 ~`) generates the output, which is then piped using the pipe operator (|) to the second command (`tr [:lower:] [:upper:]`), which converts the input from lowercase to uppercase.

Note Some command pipelines are very long: to improve readability, you can split the command across multiple lines. To do this, you use the backslash (\ ) character at the end of each line to indicate that the command continues to the next line.

The ability to efficiently use command-line utilities, combining them into effective pipelines, is fundamental for scripting, and so also for building secure, automated pipelines (DevOps and DevSecOps workflows rely heavily on composing these utilities into processing chains).

Subshells

Launching subshells enables you to create a separate execution environment by spawning a child process to execute commands in a new isolated environment. When using pipes, each command in the pipeline runs in its own subshell, but it is possible to explicitly ask to run a statement inside a subshell by enclosing it between parentheses `()` or by using certain constructs like command substitution.

An example of command substitution is

```
size=$(df -Ph /var/log | awk 'NR == 2{print $3}') \
&& echo Size is ${size}
```

The statement between the `$( )` runs in a subshell: the output is then assigned to the `size` variable in the parent shell. Then, the `echo` statement runs, printing the contents of the `size` variable.

Subshells have of course caveats and pitfalls: despite being handy for grouping commands or command substitution, their use for isolating variables is a two-edged sword you must be wary about; sometimes you may desire the outer shell not to be affected by changes to variables in the child shell, but other times you may instead require it. You must also be wary of the performance impact: opening a subshell means spawning a child process, resulting in more resource footprints. As a rule, you must avoid using subshells unless they are necessary.

The pipefail Option

The default behavior of the shell is to set the exit status of a pipeline to be the same as the exit status of the last command in the pipeline. Anyway, sometimes this behavior is not suitable; consider the previous example, this time with a non-existent path:

```
size=$(df -Ph /non-exists | awk 'NR == 2{print $3}') \
&& echo Size is ${size}
```

Despite the `df` command in the subshell failing and so returning `1`, `awk` returns a `0`: this makes the whole subshell return `0`, causing the parent shell to go on and print an empty size.

Luckily this behavior can be altered by setting the `pipefail` option: when it is set, when dealing with pipelines, the exit status of the pipeline is the exit status of the last command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully.

We can test it as follows:

```
set -o pipefail; df -Ph /non-exists | awk 'NR == 2{print $3}'
echo $?
```

This outcome of the echo command is `1` indeed. So, a safer and more robust statement is

```
size=$(set -o pipefail; df -Ph /non-exists | awk 'NR == 2{print $3}')
&& echo Size is ${size}
```

Note There are several interesting shell options that can be set to alter its default behavior, such as the `nounset` option, that causes the execution to fail immediately when trying to expand an undefined variable. For the full list of options (they depend on the shell), refer to the related man page – for example, `man bash`.

Input and Output Redirection

The shell provides input and output redirect operators, which are used to control the flow of data.

The output redirect operator > redirects the output of a command to a file. For instance, the command

```
ls -1 ~ > ~/dir-contents.txt
```

redirects the output of `ls -1`, which lists the contents of the user's home directory (`~`), to the file `~/dir-contents.txt`.

It is worthwhile to check the contents of `~/dir-contents.txt` using the command

```
cat ~/dir-contents.txt
```

As you can see, even though the redirect operation appears at the end of the command, the `~/dir-contents.txt` file already exists when `ls` is executed.

This is because the shell opens the file descriptors for the redirected file before running any commands.

Note It is important to note that the output redirect operator > opens the target file for writing, which means that each time it is used, it overwrites the existing contents and starts writing from scratch. If you want to append contents instead, you should use the output concatenation operator >>.

The input redirect operator < takes input from a file rather than from the keyboard. For example, the command

```
sort < /etc/passwd
```

reads the contents of the `/etc/passwd` file and passes it to the `sort` command, resulting in sorted output.

To see the original contents of the `/etc/passwd` file, you can use the command

```
cat /etc/passwd
```

Streams and File Descriptors

By default, a shell opens the three streams in Table 5-2.

Table 5-2. *Streams*

Stream	File Descriptor	Access	Description
Standard Input	0	Read-only	Often called *stdin* for short, it is the input stream where a program reads its data (typically from the keyboard).
Standard Output	1	Write-only	Often called *stdout* for short, it is the output stream where a program writes its results (usually the terminal).
Standard Error	2	Write-only	Often called *stderr* for short, it is the output stream where a program writes its error messages (usually the terminal).

The best practice is to print messages to the correct stream (`stdout` or `stderr`), to be able to filter them, when necessary, in shell pipelines or scripts.

For example, if you run as a regular user

```
find /root > /dev/null
```

it prints only the error message (remember that a regular user has no access to the `/root` directory) – this is because the `stdout` has been redirected to `/dev/null`. Same way,

```
find /root 2> /dev/null
```

it suppresses the error message since the `stderr` has been redirected to `/dev/null`, printing only the messages from the `stdout`.

It is of course possible to redirect the output to a specific file descriptor by using the redirect operator `>` followed by the file descriptor number prepended by `&`. For example, to write to stderr

```
echo "this message goes to stderr" >&2
```

If necessary, it is even possible to open custom file descriptors – for example:

```
exec 3> mylogfile
```

opens the file `mylogfile` for writing and assigns to it the file descriptor `3`. We can then redirect text to that file descriptor the same way we did in the previous example, this time using `&3`:

```
echo "1st message to log" >&3
echo "2nd message to log" >&3
echo "3rd message to log" >&3
```

When it is no longer necessary to use that file descriptor, we can close it as follows:

```
exec 3>&-
```

If needed, we can even of course open a file descriptor for read access:

```
exec 4< mylogfile
```

Once opened, we can read a single line storing the contents into a variable. For example, to read a line from the `4` file descriptor, storing its contents into the `line` variable

```
read line <&4
```

we can then view the contents by typing

```
echo "${line}"
```

This means that to read the whole file it is necessary to put it inside a loop, for example:

```
while IFS= read -r line <&4; do
    echo "Read line: ${line}"
done
```

As you can see, since we have already read the first line, the output of the above statement starts with the second line. Since we don't need it any longer, let's close the `4` file descriptor:

```
exec 4>&-
```

Note The `IFS` variable defines the Internal Field Separator: since its default value is the space character, to read a whole line, we must set it as empty to avoid the read stopping at the first space character, skipping the rest of the line.

Here-String

A here-string (<<<) provides a convenient alternative to using pipes (|) to provide a string as input to a command directly from the shell. For example:

```
wc -w <<< "these are four words"
```

generates the same the outcome of

```
echo "these are four words" | wc -w
```

However, it's important to note that under the hood, they work in a very different way: when using a pipe, the data flows through a kernel-managed buffer, and the pipe is represented in the /proc filesystem; a here-string instead creates a temporary file (or a file descriptor) that contains the provided string. Since this file is typically created in memory, this file is not linked to the filesystem.

Note The rule of thumb for deciding which one to use is that a pipe is a mechanism for inter-process communication that allows the output of one command to be used as the input for another. A here-string instead is a way to provide a string as input to a command directly from the shell. So, for example, while it is technically possible to write a statement such as `sort -t: -nrk3 <<< "$(grep "ss" /etc/passwd)"`, it is an improper use of the here-string leading to poor performance due to the use of a subshell – the command within `$(` and `)` runs in a separate shell. A better approach would be to use a pipe: `grep "ss" /etc/passwd | sort -t: -nrk3`.

Heredoc

Heredoc provides a convenient way to supply multiline input to programs that can read from the `stdin`. The syntax to start the heredoc is the << operator followed by the marker you want to use as a delimiter. A real-life example is

```
mail -s "Subject" marco.carcano@carcano.corp <<EOF
Hi dear,
I found a good book for learning devsecops pillars on Linux.
Cheers
EOF
```

As you can see, the multiline input containing the actual mail message is passed to the *mail* command-line utility by using a heredoc delimited by the `EOF` marker.

Note It is of course mandatory to choose as delimiter something that is not contained in the middle of the multiline text.

Heredoc is often used to create a whole document on the fly using `cat`: the trick is launching the `cat` command-line utility by redirecting its output to a file using the redirect operator `>` and reading the text using the heredoc.

For example:

```
cat << EOF > greet.sh
echo "Hi ${USER}"
EOF
```

creates the `greet.sh` script that greets the user launching it, … or not? Well, technically it greets but, since variables by default get expanded in heredocs, the `${USER}` gets expanded in the user who generated the script using the heredoc statement, causing the script to always greet that user and not the one launching it.

You can prevent variable expansion by prepending a backslash character (`\`), for example:

```
cat << EOF > greet.sh
echo "Hi \${USER}: this script was created by ${USER}"
EOF
```

When I tested this statement, I was logged as `vagrant` user, so the outcome was generating the `greet.sh` containing:

```
echo "Hi ${USER}: this script was created by vagrant"
```

As you can see, the first USER variable was not expanded, allowing the script to cheer whoever runs it.

Note If necessary, it is possible to prevent the expansion of every variable in the heredoc by prepending the backslash character (\ ) when declaring the EOF marker – for example: `cat << \EOF > greet.sh`.

Please be wary that the above example using the output redirection operator > has the shortcoming of not being usable when the current user does not have permissions for opening the destination file for writing: indeed, even when using sudo, the file where the cat's output is redirected gets opened in the calling shell that is running as the current user.

The trick to overcome this problem is piping the cat command-line utility's output to a sudo statement running the tee command-line utility. For example:

```
cat <<EOF | sudo tee /etc/sysctl.d/net.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF
```

Since in that statement tee runs wrapped by sudo, it runs as the *root* user: this means it has the rights to write the file.

Note For the sake of completeness, the tee command-line utility prints the input passed to it to the standard output and to the specified file overwriting its contents: if you need to append contents, just specify the append option (-a) to tee.

Before moving further, remove the file we just created:

```
sudo rm -f /etc/sysctl.d/net.conf
```

Don't forget this because it may interfere with the labs in the rest of the book.

Note As you can easily guess, heredocs are useful not only in shell scripts, but also when writing documentation, such as when documenting a procedure – the reader can just cut and paste the text from the documentation.

Xargs

It is a command-line utility that constructs and executes command lines by reading items from standard input or from a file. By default, it runs commands one at a time, but it can also execute commands in parallel by using the `-P` option followed by the number of instances to run concurrently. One of its most useful features is the fine-grained control it provides for converting input into usable arguments for the command, allowing the use of placeholders to position arguments correctly, especially when dealing with complex command syntax.

As an example use case, you can create a file containing a list of filenames by running the following commands:

```
echo "file1.txt" > files.txt
echo "file2.txt" >> files.txt
echo "file3.txt" >> files.txt
```

Let's take a look to the outcome:

```
cat files.txt
```

Now, we can read the file contents and pipe it to `xargs` as follows:

```
cat files.txt | xargs touch
```

In this case, `xargs` combines all the lines from the input into a list of arguments and launches the specified command once, passing the list of arguments to it. The outcome is the same as if we were running `touch file1.txt file2.txt file3.txt`.

Some commands like `curl` expect only a single argument, so the default `xargs` behavior of building and passing the list of arguments causes syntax errors.

In such a scenario, by passing the `-n 1` option, it is possible to have `xargs` repeat the command with a single argument as many times as there are the arguments in the list, passing each time the next argument.

As an example, consider you have a file named `components.lst` containing URLs (one per line) that you need to download, you can use the `-n 1` option with `xargs`, as shown by the following example:

```
xargs -a components.lst -n 1 curl -O
```

Sometimes the argument must be inserted in a specific spot of the statement to run. In this case, simply add `-I{}` as an `xargs` argument and put the `{}` placeholder in the spot where the argument is supposed to be.

A very typical use case is renaming files, for example:

```
ls *.jpg | xargs -I{} mv {} image_{}
```

renames every file with a filename ending with `.jpg` adding `image_` as prefix.

An advanced use case is when having to deal with pipes in the statement run by `xargs`: as an example, suppose the `components.lst` file contains URLs to download gzipped tarballs and that we need to extract them on the fly to the already existent `~/target-directory`.

Your first attempt probably would be something such as

```
xargs -a components.lst -I{} curl -s -L {} -O \
| tar xvfz - -C ~/target-directory
```

Sadly, it won't work – it will extract the contents only of the first downloaded file. The problem is that the pipe to the tar command-line utility must run for every single statement generated by `xargs`.

The trick in this case is to spawn a subshell such as

```
xargs -a components.lst -I{} sh -c \
"curl -s -L {} | tar xvfz - -C ~/target-directory"
```

Scripting Concepts

In this section we explain the fundamental concepts mostly related to scripts.

Execution Permission

A script can run either directly or indirectly, by having a shell loading its file and then interpreting the contents.

To better understand this concept, create the `hello.sh` script by running the following `heredoc`:

```
cat << \EOF > hello.sh
echo "Howdy ${USER}"
EOF
```

Because of the default `umask`, the `hello.sh` file is created without the POSIX execute permission set. We can easily verify this by running `ls -l hello.sh`:

```
-rw-r--r--. 1 vagrant vagrant 68 Dec 26 11:35 hello.sh
```

If we try to directly launch the script by running `./hello.sh`, we indeed get a permission denied error.

```
-bash: ./hello.sh: Permission denied
```

It is instead possible to indirectly run the script by invoking a shell and having it parsing it.

For example:

```
bash hello.sh
```

This works because from the system perspective, the launched program is the shell: once launched, a program can execute whatever it needs, including statements loaded from an external file.

To try a direct run, we must first set the POSIX execution permission as follows:

```
chmod +x hello.sh
```

Let's now try running it:

```
./hello.sh
```

The script file contents are executed by the current shell, interpreting the statements inside it.

The Shebang

The `hello.sh` script is still missing a crucial element that can lead to portability issues: the shebang, which stands for "shell bang." This special statement specifies the interpreter that must be used to execute the rest of the script.

Without a shebang, the script file contents are executed by the current shell, interpreting the statements inside it.

Note Mind that, besides BASH, there are many other shells available, such as the C shell, the Z shell, and so on – BASH is just the most popular one.

Let's clarify this concept and the shortcomings addressed by using the shebang with an example.

First, create the `conf.cfg` configuration file as follows:

```
cat << EOF > conf.cfg
SUPPORT_TEAM=DevOps
SUPPORT_EMAIL=devops@carcano.corp
EOF
```

Then, create the actual script:

```
cat << \EOF > key-value.sh
source conf.cfg

echo "Team: ${SUPPORT_TEAM}"
echo "Email: ${SUPPORT_EMAIL}"
EOF
```

Finally, assign execute permissions:

```
chmod +x key-value.sh
```

This script has been designed to run on a BASH – let's try running it from the current shell:

```
./key-value.sh
```

Since BASH is the default shell in most of the Linux distributions, this run of the script worked without any issue.

The obvious shortcoming is that the script will behave differently if it is run by a shell that is not the one it was designed for.

It is quite easy to simulate a scenario where this shortcoming arises – just switch to the sh shell by typing

```
sh
```

then run again the script:

```
./key-value.sh
```

The outcome is

```
./key-value.sh: line 1: source: conf: file not found
```

The above error is not caused by wrong script statements: it is due to the use of a wrong shell interpreter.

Note As you can see, besides syntax differences, every shell has its own nuances. In this example, we hit the difference in the implementation of the `source` command: in BASH it looks for the file in the current working directory, whereas in `sh` it looks in the application's directory (`/usr/bin`).

Adding the shebang addresses this problem, telling the shell which program to run for processing the next statements.

The shebang is placed on the first line of the script as a comment. It begins with the # character, is immediately followed by the ! character, and then includes the absolute path to the interpreter that will execute the script.

Just add `#!/bin/bash` as the first line of the `key-value.sh` script so that it looks as follows:

```
#!/bin/bash

source conf.cfg
echo "Team: ${SUPPORT_TEAM}"
echo "Email: ${SUPPORT_EMAIL}"
```

After adding the shebang, it is now possible to run the script from any shell: as a test, try running it again within the sh session you just opened: this time it should work without any issue.

Once done, exit the session back to the BASH by typing exit.

Examples of shebang are

```
#!/bin/sh
#!/usr/bin/csh
#!/usr/bin/zsh
#!/usr/bin/python3
```

As you can see from the last shebang that is used to switch to Python 3, the shebang is not used only for switching to the proper shell: it can specify any application.

Note To maximize portability, the best practice is to use #!/usr/bin/env followed by the script interpreter to be invoked, such as bash, sh, or python. For example, #!/usr/bin/env python3. Please mind anyway that this requires the invoked application to be already installed and available in the PATH environment variable.

The Special LC_ALL=C Locale

The system locales are a set of variables that contain all the information required to elaborate data in compliance with the formats of the region of the world the user wants to use. By default, they are often set to the region the user lives in, but they can be set to whatever region the user prefers.

This set of variables can be listed as follows:

```
locale
```

Some of the variables, such as LC_NUMERIC and LC_CTYPE, affect numeric values, such as decimal and thousands separator, and which characters are considered letters, digits, or punctuation.

Since these values change according to the locale, when scripting the best practice is to set `LC_ALL=C`: this way all the locale information is set in a predictable way, not only in terms of numeric formats, but also in terms of sorting and collation. In addition to that, many times the setting `LC_ALL=C` performs even better than any other setting.

Remember to always export the `LC_ALL` variable, to have it set also in every subshell:

```
export LC_ALL=C
```

Writing Comments

Every good software, no matter if it is a complete application or a simple script, must contain comments along with the source code. Writing effective comments can really improve software maintainability, helping other developers to quickly make sense of the code.

Please mind that it is pointless to describe the code – you are not writing a how-to for other developers: they are skilled enough to understand the code you wrote so the comments would only be redundant and just increase the file size.

Writing effective comments just require putting them in specific spots:

- **At the beginning of the file**: To describe the purpose of the program/script or library using very few words, explaining the covered use cases. The best practice is also to put the version of the application/script and a reference to the repository where it is possible to find the source code. Do not put here information that would be anyway available in the software repository: they would bring no real benefit, since they will quickly become outdated: the reference to the software repository is instead an effective information, since it tells where to quickly find current and accurate information.
- **Right before any function's definition, any object, and any method**: Add a short summary information of its purpose, describing both input and output parameters – if the language does not enforce type checks, the best practice is also to specify the expected type of each parameter (integer, float, string, or object type).

- **Close to the lines of complex business logic**: Sometimes it is necessary to write small portions of business logic that, despite being effective, rely on an overall vision of the developed solution possessed only by who wrote them. Since overall vision tends to become opaque and slowly disappears after some time after the release, these small portions of code can become a maintenance trap. Even worse, it is very unlikely that the person having to rework them will have the overall vision too. For this reason, when writing portions of code having this maintainability requirement, it is very important to add some lines of comments providing context, rationale, or explanations that are not immediately obvious from the code itself.
- **Near flaky code lines**: When writing a workaround to fix an emergency, it is quite common to write flaky code aimed at quickly restoring the functionality, postponing to a less stressful time the writing of the definitive fix. In such a scenario, it is wise to add some comment with prefixes such as `FIXME` or `TODO` to mark the spot of the code still needing a rework or still pending to be implemented, shortly describing them.

When working with BASH, any content following the # character is considered a line comment. So, for example:

```
if [[ -n ${VAR+x} ]]; then
  # code snippet not working with associative arrays
  echo "VAR is set: value is ${VAR}" # not working with arrays
fi
```

Languages such as Python support the `docstring` format: when dealing with development languages supporting docstrings, the best practice is to write comments in compliance with that format, since it enables the automatic generation of the documentation.

For example, it is possible to generate the documentation of Python code containing docstring-compliant comments by running `pydoc` followed by the path to the python script. For example:

```
pydoc myscript.py
```

Static Analysis Tools

Static analysis tools provide a very effective way to write good quality programs. Besides performing a syntax check, they also perform style check, highlighting the lines of code with wrong or non-compliant syntax. They also provide guidelines to resolve the non-compliant lines of code.

Checking the code with these tools strongly helps in writing good code, mitigating the risk of unpredictable outcomes that can be caused by poorly written code.

When dealing with BASH, the most used static analysis tool is `shellcheck` – you can easily install it by running

```
sudo dnf install -y shellcheck
```

Once installed, you can check your script by simply running `shellcheck` followed by the path to the script you want to check.

For example:

```
shellcheck hello.sh
```

As you can see, it is shouting at us because it does not contain the `shebang`: add the `shebang`, for example, `#!/bin/sh`, re-run the shellcheck statement, and the error will disappear.

Note Mind that `shellcheck` is available as a plugin for several IDEs including Visual Studio Code.

Profiling Using the Time Utility

The `time` command-line utility is used to measure the duration of execution of a command, providing also resource usage statistics. It is particularly useful to check the performance of your pipelines and script: you just must run the pipeline (or the script/application) you want to monitor performances prepending the word `time`.

For example:

```
time grep "ss" /etc/passwd | sort -t: -nrk3
```

When the run completes, `time` prints the statistics with the real time, the user time, and the system time spent during the run:

```
real    0m0.014s
user    0m0.005s
sys     0m0.019s
```

Note If necessary, it is possible to use a few variables to customize the time's output. For example: `export TIMEFORMAT=$'\nReal=%3lR\nUser=%3lU\nSys=%3lS'`.

BASH Cheat Sheet

This section is a small cheat sheet you can use as a quick reference when writing BASH scripts. It contains snippets not only with the current syntax, but also the legacy one: the aim is to equip you with the necessary skills when writing new scripts as well as helping you to understand the existing ones (at your workplace it won't be so difficult to stumble upon very old ones too using legacy syntax).

Of course, to leave enough space for all the other topics of this book, we cannot thoroughly delve into everything, but the concepts explained in this section are more than enough to help you become proficient in your daily work.

Note I strongly suggest you have a look and bookmark the Devhints' BASH scripting cheat sheet – https://devhints.io/bash.

Script Positional Arguments

When launching shell scripts, it is possible to pass values to them using positional parameters as shown below:

```
./foo.sh arg1 arg2 argN
```

The above statement launches the `foo.sh` script in the current working directory passing `arg1` as first argument, `arg2` as second argument, and so on.

Table 5-3 summarizes the special variables that can be used to access the positional argument's value inside the script.

Table 5-3. *BASH: script positional argument special variables*

Example	Description
`${0}`	The script's filename.
`${1}` `${2} .. ${n}`	Positional arguments – `${1}` is the first one, `${2}` is the second one, and so on.
`"$*"`	When quoted as `"$*"`, it expands to a single string with all positional parameters concatenated together, separated by the first character of the `IFS`.
`"$@"`	When quoted as `"$@"`, it expands each command-line argument (like $1, $2, etc.) into its own separate, quoted string. This keeps spaces inside each argument intact and treats them as individual items, similar to how you'd list elements in an array.
`${#}`	The number of positional arguments passed to the script.

Variables

Inside a script you can manage values by using variables: they can be either environment variables set when launching the script, for example:

```
BAR=waldo; ./foo.sh
```

Or previously exported variables:

```
export BAR=waldo
./foo.sh
```

In addition to that, you can set custom variables. Table 5-4 summarizes the basic operations with variables.

Table 5-4. *BASH: basic operations with variables*

Example	Description
`declare -i NUMBER=1`	Assign `1` to the variable `NUMBER` of integer type. Declaring by setting a type enforces numeric operations. Mind that internally BASH still treats it as a string.
`GREET="hi dear"`	Assign the `hi dear` string as the value of the variable `GREET`.
`echo "${GREET}"`	Print the value of the variable `GREET`.
`unset GREET`	Unset (remove) the variable `GREET`.

Arrays

Arrays are variables that can hold multiple values indexed by numeric keys: you can access elements using their index.

Table 5-5 summarizes how to operate with arrays in the BASH.

Table 5-5. *BASH: basic operations with arrays*

Example	Description
`declare -a list=("foo" "bar" "baz")`	Declare the `list` array with elements `foo`, `bar`, and `baz`.
`echo ${#list[@]}`	Print the array's length.
`echo "${list[@]}"`	Print every value in the array – in this example, it prints `foo bar baz`.
`echo "${!list[@]}"`	Print every index in the array – in this example, it prints `0 1 2`.
`echo "${list[2]}"`	Print the value of the item with index `2` – `baz` in this case.
`list[3]="waldo"`	Add the element `waldo`.
`unset "list[3]"`	Unset (remove) the fourth element – the one with index `3`.

BASH supports also associative arrays: these extend the concept of array, using string keys instead of numeric indices, functioning like dictionaries. This means that you can access their values using the associated keys.

Table 5-6 summarizes how to operate with associative arrays in the BASH.

Table 5-6. *BASH: basic operations with associative arrays*

Example	Description
`declare -A dictionary`	Declare the `dictionary` associative array.
`dictionary[name]="John"` `dictionary[surname]="Doe"`	Assign `"John"` to the dictionary item `name` and `"Doe"` to the dictionary item `surname`.
`echo "${dictionary[@]}"`	Print every value in the array – in this example, it prints `John Doe`.
`echo "${!dictionary[@]}"`	Print every key in the array – in this example, it prints `surname name`.
`unset "dictionary[name]"`	Unset (remove) the item having `name` as key.

Special Variables

Scripts run a lot of programs within them: to help deal with managing their execution and controlling their runtime status, the shell provides some special variables.

The most used ones are listed in Table 5-7.

Table 5-7. *BASH: process-related special variables*

Example	Description
`${?}`	Return value of the last exited process
`${!}`	Last background Process ID
`${$}`	Process ID of the current shell

Test If a Variable Is Set

It is quite common having to check if a variable is set – the following snippet addresses this specific use case:

```
if [[ -n ${VAR+x} ]]; then
  echo "VAR is set and its value is '${VAR}'"
fi
```

Please mind that if VAR is an array, you must use [@] to expand all its values – for example, `echo "VAR '${VAR[@]}'"`. In addition to that, mind that this test does not work if VAR is an associative array.

Note It is mandatory to use double square brackets [[...]] for this type of conditional expression, since the legacy one (single square brackets [...]) does not support this syntax and may not work as expected or result in a syntax error.

Empty, Not Empty, and Length

Table 5-8 summarizes how to test if a variable is empty, if it is not empty, and how to return its length.

Table 5-8. *BASH: variables checking empty, not empty, and length*

Example	Description
`[[ -z ${VAR} ]] && echo "empty"`	Print the word `empty` if the VAR variable is zero-length (meaning empty). Be wary that `-z` is not string specific, and so it does count spaces in the computation – for example, if the value of VAR is `"  "`, then `-z` returns `1`.
`[[ -n ${VAR} ]] && echo "not empty"`	Print `not empty` if the VAR string is not empty. Be wary that `-n` is not string specific, and so it does also count spaces in the computation – for example, if the value of VAR is `"  "`, then `-n` returns `1`.
`echo ${#VAR}`	Print the length in characters (or digits) of the VAR variable.

Numbers

This section summarizes how to deal with variables containing numeric values in BASH. For completeness, note that BASH treats all variables as strings by default. To perform arithmetic operations, BASH temporarily switches to a numeric context, converting the string values into integers as needed. Statements that require processing in numeric context are enclosed in double parentheses `(( ... ))`. Since BASH uses integers in this numeric context, it does not support floating-point arithmetic, which can lead to truncation of decimal values.

Increment and Decrement

Similarly to C/C++, it is possible to use increment and decrement operators.

For example:

```
declare -i COUNTER=42
((COUNTER++))  # outcome is 43
((COUNTER--))  # outcome is 42
((COUNTER+=24)) # outcome is 66
((COUNTER-=3)) # outcome is 63
((COUNTER*=2)) # outcome is 126
((COUNTER/=3)) # outcome is 42
((COUNTER%=9)) # outcome is 6
```

Math Operations

Math operations in BASH are performed using arithmetic expansion. You can enclose the expression in double parentheses, as shown in the following example:

```
A=7
B=2
SUM=$((A + B))        # outcome is 9
DIFFERENCE=$((A - B)) # outcome is 5
MULTIPLY=$((A * B))   # outcome is 14
MODULO=$((A % B))     # outcome is 1
DIVIDE=$((A / B))     # outcome is 3
```

Note Especially in old scripts, you can find math operations using the *let* operator. For example: `let SUM=A+B`. While *let* is still valid and works for arithmetic operations, many BASH users prefer using the `$((...))` syntax for arithmetic because it is more versatile and clearer.

Although it may look odd having `3` as the outcome of `7 / 2`, this instead should be expected: since as we anticipated BASH does not support floating-point numbers, then the outcome is just truncated as an integer.

To overcome this limitation, when dealing with floating-point arithmetic, you can use the `bc` command-line utility.

For example:

```
echo "${A}/${B}"| bc -l
prints 3.50000000000000000000
```

Note Mind that, depending on the installation class you used when you installed the Linux distribution, `bc` may not be installed by default – if you need it, you must install it by running `sudo dnf install -y bc`.

If you need to limit the result to only two decimal digits, you can specify the `scale` parameter as in the below example:

```
echo "scale=2; ${A}/${B}"| bc -l #3.50
```

Rounding

You can round values using the `printf` command. Table 5-9 shows some examples along with the outcomes.

Table 5-9. *BASH: rounding values using printf*

Statement	Outcome
`printf '%.*f\n' 0 6.66`	`7`
`printf '%.*f\n' 1 6.66`	`6.7`
`printf '%.*f\n' 2 6.66`	`6.66`
`printf '%.*f\n' 3 6.66`	`6.660`
`printf '%.*f\n' 3 6.666`	`6.666`
`printf '%.*f\n' 3 6.6666`	`6.667`

Strings

This section summarizes the most common operations you may encounter when having to deal with variables containing string values.

Counting the Number of Words

The following snippet shows how to count the number of words into a string variable:

```
STR="Hello world"
echo "${STR}" | wc -w
```

Manipulating String Contents

Table 5-10 summarizes how to manipulate string contents.

Table 5-10. *BASH: manipulating string contents*

Example	Description
`echo "${STR// /}"`	Remove the spaces – an alternate version using `tr` is `echo "${STR}" | tr -d " "`
`echo "${STR^^}"`	Convert the value of the `STR` string to upper case – an alternate version using `tr` is `echo "${STR}" | tr '[:lower:]' '[:upper:]'`
`echo "${STR,,}"`	Convert the value of the `STR` string to lower case – an alternate version using `tr` is `echo "${STR}" | tr '[:upper:]' '[:lower:]'`

Extracting a Substring

It is possible to extract a substring from a string by expanding it as follows:

```
echo -n "${STR:offset:length}"
```

The above example prints the substring starting at the specified `offset` position, getting the number of characters specified by `length`.

Another use case is extracting only the last part of a string. For example, to print the last five characters of the `STR` string:

```
echo -n "${STR: -5}"
```

Checking Matching Contents

Given the `EMAIL="marco.carcano@carcano.corp"` string, the following snippets show how to check if a string

- Contains `rco`:

  ```
  [[ "${EMAIL}" == *rco* ]] && echo "matches"
  ```

- Is an email – in this case, we use a regular expression (RegEx):

  ```
  [[ "${EMAIL}" =~ ^.+@.+\..+$ ]] && echo "matches"
  ```

Note To keep it simple, we used quite a flaky and simple regular expression. We will discuss regular expressions in a Chapter 7 – for now just get that BASH supports them.

Replacing Contents

The following snippets show how to replace the `pattern` token with the value `replacement` in the `STR` string:

```
echo "${STR/pattern/replacement}"
```

if you want to replace every occurrence:

```
echo "${STR//pattern/replacement}"
```

Globbing

When dealing with filenames and directories, the BASH does not support regular expressions but implements globbing.

Globbing is the process of expanding filenames accordingly to the rules shown in Table 5-11.

Table 5-11. *Globbing*

Stream	Description
?	Matches any character, but only once. You can use it multiple times, for example, ??? matches three characters in length
*****	Matches zero or more words (in this sense word means sequence of characters)
[]	Define a range of matching characters, for example: • `[ab]` matches characters a or character b • `[a-c]` matches characters from a to c

For example:

```
ls -d /usr/share/doc/[a-c]*
```

You can specify more than one globbing pattern by using braces – for example:

```
ls -d /usr/share/doc/{a*,d*}
```

List and Sequences

It is possible to specify a list of items by enclosing its items into braces. As an example, let's create the directory tree for the fancy `foo` application:

```
mkdir -p -m 755 foo/{etc,bin,lib,log}
```

This creates in the current working directory the following directories:

```
foo/etc
foo/bin
foo/lib
foo/log
```

Note Structuring an application within a self-contained directory tree beneath the user's home directory tree is considered a best practice in Linux, since it provides a tiny and convenient structure where the user has write access without the need of additional grants. In this basic example, `etc` is the directory for configuration files, `bin` is for executable files, `lib` is for libraries, and `log` is for log files. Using such a structure enables any user without administrative privileges to run the application. To support system-wide deployments that adhere to the Linux filesystem hierarchy, it is essential to develop the application in a way that allows it to specify custom paths for these directories, such as loading configuration files from `/etc/foo` instead of just from the `etc` directory. Applications developed using the `Autotools` framework typically support via configure scripts specifying options like *--prefix*, *--sysconfdir*, *--libdir*, etc., to specify custom directories for each specific purpose during build/installation.

In the same way, it is possible to generate a sequence of items by enclosing its definition into braces.

The syntax of the sequence's definition consists of providing the start item and the end item separated by two periods (`..`) – for example:

```
echo {0..12}
```

prints

```
0 1 2 3 4 5 6 7 8 9 10 11 12
```

Other examples of sequences are

- `{3..-2}` expanded into `3 2 1 0 -1 -2`
- `{a..g}` expanded into `a b c d e f g`
- `{g..a}` expanded into `g f e d c b a`

User Input

Sometimes it is needed to write interactive scripts: this requires interrupting the flow waiting for the user's input.

This section provides a few snippets for the most common use cases.

The Read Command

The following snippet prints the `Enter your name:` message and waits for the user to type something, ending the process when the user presses the *enter* key. The input is then stored in the `name` variable.

```
read -r -p "Enter your name: " name
```

In the same example, but this time disabling echoing the typed text (useful when entering sensitive data such as passwords)

```
read -r -s -p "Enter your password: " password
```

The Select Command

The `select` command enables the user to pick a value from a list: the advantage is that it spares us from typing the value, avoiding the risk of typos.

This small code snippet lists the `/etc` subdirectories, asking the user to pick one.

```
     mapfile -t DIRS < <(ls -d /etc/*)
PS3="Pick a directory: "
while true; do
    select DIR in "${DIRS[@]}"; do
        if [[ -n "$DIR" ]]; then
            break 2
        else
            PS3="Invalid choice - Pick a directory: "
            break
        fi
    done
done
echo "You chose ${DIR}"
```

The list is fetched using the `ls` command in a subshell, using the output to feed the `mapfile` command to read it into the `DIRS` array. It then uses the `select` command to create a simple menu from which to choose one of the directories. Since the `select` prompt is controlled by the `PS3` special variable, we also set it with a convenient message.

The loop exists only after a valid selection has been performed.

Testing Files and Directories

When writing automation scripts, a very common need is checking files' and directories' existence. This section shows you how to deal with the most common use cases.

File Existence

This snippet checks if the `/var/log/messages` file exists.

```
if [[ -f /var/log/messages ]]; then
  echo "exists"
fi
```

Other common file existence tests parameters are

- `-s`: Ensure it exists and has a size greater than zero (it is not empty).
- `-r`: Ensure it exists and the user running the statement has read access on it.
- `-w`: Ensure it exists and the user running the statement has write access on it.
- `-x`: Ensure it exists and the user running the statement has execution access on it.

Directory Existence

This snippet checks if the `/var/log` directory exists.

```
if [[ -d /var/log ]]; then
  echo "exists"
fi
```

Flow Control

This section contains a few snippets showing how to use flow control blocks.

If-Elif-Else

This snippet shows how to use an `if-elif-else` block to process the argument passed to the script, printing a message indicating the types of characters (numbers, letters, symbols) contained in the argument:

```
#!/bin/bash
if [[ -n ${1+x} ]]; then
   if [[ -z "${1//[[:digit:]]}" ]]; then
     echo "${1} is a number"
   elif [[ -z "${1//[[:alpha:]]}" ]]; then
     echo "${1} is one or more characters"
   elif [[ -z "${1//[[:alnum:]]}" ]]; then
     echo "${1} contains both numbers and characters"
   else
     echo "${1} can contain both numbers, characters and punctuation"
   fi
else
  echo "The script requires an argument" >&2
  exit 1
fi
```

The above script uses the extended test command `[[ ... ]]`.

Table 5-12 summarizes the comparison operators that can be used with test `[ ... ]` and extended test `[[ ... ]]` commands.

Table 5-12. *BASH: comparison operators for test and extended test commands*

Example	Description
`STR == "foo"`	The value of the `STR` string is `foo`.
`STR != "foo"`	The value of the `STR` string is not `foo`.
`STR =~ regexp`	Check the value of the `STR` string using the extended regular expression (ERE) `regexp`.
`NUMBER -eq 10`	The value of the `NUMBER` variable is `10`.
`NUMBER -ne 10`	The value of the `NUMBER` variable is not `10`.
`NUMBER -lt 10`	The value of the `NUMBER` variable is lower than `10`.
`NUMBER -le 10`	The value of the `NUMBER` variable is lower or equal to `10`.
`NUMBER -gt 10`	The value of the `NUMBER` variable is greater than `10`.
`NUMBER -ge 10`	The value of the `NUMBER` variable is greater or equal to `10`.

Note When dealing with numeric values, instead of using the test or extended test commands, you can switch to the numeric context and write conditions using `(( ... ))`. This allows you to use expressions such as `(( NUMBER == 10 ))`, `(( NUMBER != 10 ))`, `(( NUMBER >= 10 ))`, and so on.

Case

This snippet shows a simple `case` block with the words `start` and `stop` as the only valid first argument values to the script:

```
case "${1,,}" in
  start)
    echo "start"
    ;;
  stop)
    echo "stop"
    ;;
  *)
```

```
    echo "Usage: ${0} {start|stop}" >&2
    exit 1
    ;;
esac
```

Note Please mind how we used the double comma to convert to lowercase the first argument `${1,,}` passed to the script.

Control Structures

This section contains a few snippets showing how to use control structures.

For Loop

It is used to repeat the block of statements enclosed between the `do` and `done` keywords.

A practical usage is iterating over a list of items. The following snippets show how to generate a list of items using a subshell command and iterate through the contents using `for`:

```
for i in $(ls)
do
  echo "item: ${i}"
done
```

Here is another example of populating the list of items using a subshell, this time invoking the subshell using a different syntax – the `seq` command-line tool is used to generate a sequence of numbers between one and ten:

```
for i in $(seq 1 10)
do
  echo "${i}"
done
```

Note I showed this only for didactical purposes: as we said, spinning up subshells negatively affects performance. We already saw a more effective way of iterating having BASH itself generating the sequence using the curly braces – for example: `echo {0..12}`.

The `for` statement can also use a C-like syntax. For example, to repeat the code block ten times:

```
for ((i=0; i<10; i++))
do
  echo "${i}"
done
```

While Loop

A `while` loop is a condition-controlled loop, meaning that the condition is evaluated before entering the loop and at the beginning of each iteration.

The condition to enter and stay in the loop is having the `while` statement evaluate as `true`.

This means

- If the condition is `false` before entering the loop, then the loop is skipped entirely
- Once inside the loop, at the first iteration when it is evaluated `false`, the execution exits the loop

The following snippet is another example of repeating the code block ten times:

```
COUNTER=0
while [ ${COUNTER} -lt 10 ]
do
  echo "counter is ${COUNTER}"
  let COUNTER=COUNTER+1
done
```

This example is an infinite loop sleeping for 30 seconds at each iteration:

```
while true
do
  sleep 30
done
```

While they may seem like a theoretical construct with no practical use, there are indeed scenarios that require an infinite loop. A common example is using it as an entry point when troubleshooting a non-starting Linux container or when you simply need to keep the container running for experimentation. In such cases, the container starts by overriding the entry point with an infinite loop. Then, a shell process can be attached to that container, allowing you to run interactive commands as needed for your specific use case.

Until

It is another condition-controlled loop: this time the condition check is `until` the check condition is `false`. This means

- If it is `true` before entering the loop, then the loop is skipped
- Once inside the loop, at the first iteration when it is evaluated `true`, the execution exits the loop

Below there is a real-life example with a snippet of code looping until the JBoss Fuse CLI can connect to the server – the loop is repeated until a connection success is returned, exiting after retrying ten times:

```
COUNTER=0
until /opt/fuse/bin/client -u admin -p ${password} "info"
do
  if (( COUNTER > 10 )); then
    echo
    exit 1
  fi
  echo -n "."
  COUNTER=$(( COUNTER + 1 ))
  sleep 5
done
echo
```

Note It is possible to break the execution of a loop (`for`, `while`, and `until`) using the `break` keyword. By using the `continue` keyword, it is instead possible to immediately continue from the next iteration, skipping the execution of the remaining logic of the current iteration.

Functions

One of the key principles in software development is the Don't Repeat Yourself (DRY) approach. According to this best practice, code should never be redundant. Another important principle is Keep It Simple, Stupid (KISS), which emphasizes that code should be kept as simple as possible to facilitate maintenance.

Functions enable us to meet both principles, since they provide a convenient way to reuse the same block of code multiple times, allowing you to call them while passing input parameters as necessary. The code inside the function processes these parameters and can return output values as needed. Additionally, by using locally scoped variables, functions help manage the scope of processed values, preventing them from interfering with other parts of the script: this is particularly important in longer scripts, where there are high chances of reusing variables names.

The following example shows how to define a function using the legacy BASH syntax:

```
greet() {
  echo "Howdy"
}
```

To call the function, simply use its name:

```
greet
```

As anticipated, it is possible to pass arguments to a function: they can then be accessed inside the function using the special variables `${1}`, `${2}`, and so on, which represent the first, second, and subsequent arguments.

Here's an example of a function that takes a single argument:

```
function greet() {
  echo "Howdy ${1}"
}
greet "Marco"
```

This second example also shows the current and better way of declaring the function, prepending the `function` keyword to the function's name.

Note You must always prepend the `function` keyword, since several IDE do not recognize functions declared in the old way.

Functions in BASH can return status values between `0` and `255` to the caller using the `return` command. However, unlike many other scripting and programming languages, BASH functions cannot return computed values directly.

To overcome this limitation, computed values are often managed using global variables. The drawback of this approach is that it can lead to an excessive number of global variables in larger projects.

To mitigate this issue, when a function calls another function, you can declare a local variable in the caller function and use it to capture the output of the called function. This allows you to offload some logic to an external function while avoiding the creation of unnecessary global variables.

The following script should help to clarify this concept:

```
#!/bin/bash
function nested() {
  echo $((${1}+RANDOM))
}

function sum() {
  echo $((${1} + ${2}))
}

function outer() {
  echo "- - - - - - - - - outer() - - - - - - - - - -"
  local nested_value
  nested_value=$(nested "${2}")
  echo "nested_value = ${nested_value}"
  RESULT=$(( nested_value - ${1}))
}
```

```
outer 40 2
echo "- - - - - - - - global part - - - - - - - - - -"
echo "RESULT=${RESULT}"
echo "nested_value=${nested_value}"
```

In this example, the main logic calls the `outer` function, which sets the `nested_value` local variable. This variable is then used to collect the outcome of the `nested` function. The last statement of the `outer` function performs a mathematical operation and stores the result in the `RESULT` global variable.

The script generates different outcomes at each run, because of the use of the `RANDOM` variable in the `nested` function.

This is the output of a run:

```
- - - - - - - - - outer() - - - - - - - - - - -
nested_value = 7425
- - - - - - - - global part - - - - - - - - - -
RESULT=7385
nested_value=
```

As you can see, the `nested_value` variable has a value only inside the `outer()` function, whereas in the main logic it is without a value. This is expected and illustrates how to avoid using global variables for managing values produced by operations within functions.

Note When dealing with very large scripts, an excessive number of global variables can lead to unexpected outcomes. This is particularly true when working with multiple external files, as it becomes easy to inadvertently reuse a global variable, altering the overall state of the script and causing unpredictable effects. This potential for confusion and error is one of the reasons why, in large projects, it is often advisable to avoid shell scripts in favor of more advanced scripting languages such as Python.

Trapping Signals

The `trap` command allows you to catch signals and execute specific commands or functions when those signals are received. This technique is typically used for cleaning up resources or handling interruptions gracefully in your scripts.

Common signals that can be intercepted include

- `SIGINT` (Interrupt)
- `SIGTERM` (Terminate)
- `SIGHUP` (Hangup)
- `SIGQUIT` (Quit)
- `SIGKILL` (Kill)
- `SIGSTOP` (Stop)
- `SIGCONT` (Continue)
- `SIGUSR1` (User-defined signal 1)
- `SIGUSR2` (User-defined signal 2)

Let's illustrate the use of the `trap` directive with the following script:

```
#!/bin/bash
function cleanup {
  if [[ ${1} -ne 0 ]]; then
    echo "Exiting with status ${1}" >&2
  fi
  rm -rf "${WORKSPACE}"
}
WORKSPACE=$(mktemp -d)
echo "script started"
trap 'cleanup $?' EXIT
if (( RANDOM % 2 == 1 )); then
  exit 2
fi
echo "script ended"
```

This script can randomly complete successfully or exit with error code `2`, running on exit the `cleanup` function in both cases.

To define the triggering of this function, it uses the `trap` command to intercept the `EXIT` signal: this way the `cleanup` function is run as soon as this signal is detected.

As its name suggests, this function performs a cleanup of the script's workspace, removing the directory specified by the `WORKSPACE` variable – this directory is created in the main part of the script, to be used for temporary storage during the script's execution. In addition to that, if the exit code set at the time of calling the function is different from `0`, the function also prints an error message to the standard error.

The main part of the script begins from the `WORKSPACE=$(mktemp -d)` statement, which creates a temporary directory and assigns its path to the `WORKSPACE` variable. The logic then simulates the two distinct outcomes of a real script (success or failure) by defining an `if` block which evaluates the `RANDOM % 2` expression. This expression can randomly produce only either `0` or `1` – this gives a roughly 50% chance of getting `1`, causing the exit `2` statement.

Summary

In this chapter, we covered the fundamentals of shell scripting, focusing on

- The most frequently used command-line tools essential for effective system management and automation
- Valuable shell techniques crucial for developing professional-grade scripts and enhancing scripting capabilities

For DevOps and DevSecOps professionals, proficiency in BASH scripting is vital for automating repetitive tasks and streamlining workflows. Scripting facilitates system administration, deployment processes, and configuration management, reducing the risk of human error, enhancing consistency, and saving time. Additionally, BASH scripting enables the integration of various tools and services, allowing for seamless orchestration of development and operational tasks. Understanding BASH also empowers professionals to troubleshoot issues effectively by creating scripts that log, monitor, and analyze system behavior. Ultimately, BASH scripting is a foundational skill that enhances a professional's ability to manage infrastructure efficiently and respond swiftly to changing requirements in dynamic environments.

PART III

Basic Data Formats

In this section, we begin exploring one of the foundational pillars of DevOps: data formats. Proficiency in data formats is essential, as everything managed by programs and scripts relies on them.

This fundamental integration aligns with the core architectural principle established by Niklaus Wirth in his seminal work: *`Algorithms + Data Structures = Programs`*.

While algorithms define the logic of a script, data structures organize the information in memory. Data formats are simply the concrete text representation used to serialize, store, and exchange these structures inside files. Consequently, engineers cannot write effective automation scripts or secure system configurations without mastering how data is structured, read, and written on the disk.

This section delivers a practical examination of foundational data formats, beginning with key-value pairs, `INI` files, and `CSV` datasets. It provides concrete examples utilizing both Bash and Python scripts for real-world processing and manipulation.

For DevOps and DevSecOps professionals, it is mandatory to know how to manage key-value, `INI`, and `CSV` data formats, reading and storing them in files as necessary: they are indeed widely used for application configuration and data interchange, particularly in data pipelines.

Proficiency in these formats, along with practical experience using BASH and Python, enables professionals to design and manage reliable automation layers and system integrations that serve as the building blocks for secure enterprise architectures.

Note Before going any further, create this **chapter's workspace directory** by running `mkdir -p ~/devsecops-pillars/ch06`, and then immediately navigate to it by running `cd ~/devsecops-pillars/ch06`.

CHAPTER 6

Key-Value Pairs

Note This paragraph's sample scripts are supposed to be stored in the *key-values* directory beneath the chapter's workspace directory: create it and change directory to it – `mkdir key-values; cd key-values`

This is one of the most basic and straightforward data formats, consisting of a key-value pair separated by a delimiter character, typically an equals sign (=). For example:

```
key=value
```

A common use case is setting an environment variable in a shell, such as

```
USERNAME=mcarcano
```

Keys must be unique within a collection to avoid conflicts, while values are typically scalars (e.g., booleans, numbers, or strings).

Note Values can also be composite, such as `JSON`, `XML`, or `YAML` documents. For very long values, they can be offloaded to a dedicated file, with the key-value pair containing the file path. The application then implements a lookup mechanism to retrieve the actual value, though this may introduce performance tradeoffs like increased I/O overhead.

A collection of multiple key-value pairs is commonly called a `dictionary` (e.g., in Python) or `associative array` (e.g., in BASH or PHP). The storage for these pairs is referred to as a key-value store (KVS).

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_6

The simplest KVS is a flat file (e.g., a `.properties` or `.env` file), but it can extend to a directory tree when using separate files for composite values. For non-trivial projects or performance-critical scenarios, key-value pairs are often stored in specialized services like NoSQL databases, optimized for fast storage, retrieval, and management. Examples include `Redis` (in-memory caching), `KeyDB` (a Redis-compatible alternative), and `etcd` (distributed for orchestration). These services support sharing data across multiple hosts, enabling high availability in distributed systems.

A basic form of KVS is a "values file," commonly used for storing settings that scripts can load.

In shell scripts, key-value pairs are typically loaded as environment variables. In other languages, such as Python, they are often loaded into standard variables or dictionary objects.

The `key-value.sh` script we played with in Chapter 5 is a good and simple example of a BASH script handling key-value pairs.

Let's re-create it in the current directory as follows:

```
cat << \EOF > key_value.sh
#!/bin/bash

source conf.cfg
echo "Team: ${SUPPORT_TEAM}"
echo "Email: ${SUPPORT_EMAIL}"
EOF
chmod +x key_value.sh
```

along with its config file:

```
cat << EOF > conf.cfg
SUPPORT_TEAM=DevOps
SUPPORT_EMAIL=devops@carcano.corp
EOF
```

Just to remind ourselves of its outcome, run it once again:

```
./key_value.sh
```

Besides SHELL scripts, in the DevOps field it is quite common to also use scripting programming languages, such as Python.

Since Python is broadly used, it is certainly worth showing how to process the same key-value configuration file using it.

The following script processes the `conf.cfg` file, implementing in Python the same logic of the `key_value.sh` BASH script.

```
#!/usr/bin/python3

import sys

conf = {}
try:
    with open("conf.cfg", mode="r",
            encoding="utf-8") as conf_fh:
        for line in conf_fh:
            key, var = line.partition("=")[::2]
            conf[key.strip()] = var.rstrip("\n")
    print(f"Team: {conf['SUPPORT_TEAM']}")
    print(f"Email: {conf['SUPPORT_EMAIL']}")
except Exception as e:
    print(e, file=sys.stderr)
    sys.exit(1)
```

Save it as `key_value.py` and assign it execution rights:

```
chmod +x key_value.py
```

Then run it, just to see it in action:

```
./key_value.py
```

This very trivial script opens the `conf.cfg` file in read-only mode into the `conf_fh` file handle. It then uses a `for` block for reading it, iterating line by line: each line is split in two tokens using the = character as separator, stripping the key from unnecessary characters. For example, it removes spaces from both sides of the word and right strips the value from unnecessary newline characters too. At each iteration, the key and value are appended to the `conf` dictionary as a new dictionary item.

Note Please note also the use of the `with` keyword: with that Python takes care of automatically closing the file descriptor as soon as the run goes outside the line of code that are in the `with` block's scope.

Python scripting is of course much more complex compared to shell scripting: this is by design and expected, since they have different purposes – shell provides basic commands and facilities to combine command-line tools into pipelines; Python is a full-featured scripting language supporting object-oriented programming and importing packages from libraries. They clearly have been designed to address different use cases. The above example reveals that when dealing with basic data formats, such as key-value pairs, in terms of code length BASH is certainly a winner, … but it's not like that for other use cases, as we are about to see shortly.

Ini

Note This paragraph's sample scripts are supposed to be stored in the `ini` directory beneath the chapter's workspace directory: create it and change directory to it – `cd ~/devsecops-pillars/ch06; mkdir ini; cd ini`.

Key-values files have the shortcoming of being one-dimensional. This limits their usage to only quite basic use cases: as soon as the number of keys with similar purposes grows, it becomes indeed harder assigning unique names as keys.

The risk of naming collisions increases indeed, quickly leading to unclean and messy naming, and even if you define a tidy naming scheme, it leads to excessively long key names.

A step forward to mitigate this shortcoming is using the INI format: INI enables listing multiple key-value pairs grouping them into dictionaries assigning them a name – often called `section` or `stanza`. Dividing into sections enables reusing the same unique keys with different values in different stanzas, since each stanza is a standalone key-value dictionary.

As an example, consider a script needing to access multiple services using different credentials – a suitable INI file could be a `credentials.ini` file with the following contents:

```
[ldap]
username=ldapsvcuser
password=mypassword

[database]
username=dbuser
password=otherpassword
```

As you can guess, this format is too complex for the shell alone, since it does not have any built-in command for managing it. This means that, to process INI-formatted files in shell scripts, it is necessary to rely on external command-line utilities such as *sed* or *awk*, processing their outcome (exit code or output) using conditionals or storing them into variables for later processing.

The username and password from the `ldap` stanza in the `credentials.ini` file can be extracted by using the following one-liner:

```
sed -n '/\[ldap\]/,/^\[/ { /\[.*\]/b; p}' credentials.ini
```

The output is as follows:

```
username=ldapsvcuser
password=mypassword
```

To see how to use this sed statement inside a shell script, create the `credentials.sh` shell script with the following contents:

```
#!/bin/bash
CREDENTIALS_FILE="credentials.ini"

if [[ -z "${1}" ]]; then
  echo -n "You must specify the stanza containing the " >&2
  echo "variables to load" >&2
  exit 1
fi

STANZA=${1}

if [[ ! -f "${CREDENTIALS_FILE}" ]]; then
  echo "${CREDENTIALS_FILE} file not found!" >&2
  exit 1
```

```
fi
if [[ ! -r "${CREDENTIALS_FILE}" ]]; then
  echo "Unable to read ${CREDENTIALS_FILE} file" >&2
  exit 1
fi
# WARNING: the below eval statement is not split on two lines
eval "$(sed -n "/\[${STANZA}\]/,/^\[/ { /\[.*\]/b; p}" "${CREDENTIALS_
FILE}")"

if [[ -z "${username}" ]] || [[ -z "${password}" ]]; then
  echo "stanza \"${STANZA}\" not found!" >&2
  exit 1
fi

echo "${STANZA} username: ${username}"
echo "${STANZA} password: ${password}"
```

Besides the `sed` one-liner, the script is quite trivial: it

- Makes sure the user provided at least the first argument while running the script
- Stores in the `STANZA` variable the first argument passed to the script
- Makes sure the `credentials.ini` file exists and the process has read permission on it
- Runs the sed one-liner within an `eval` statement, causing the output of the sed one-liner to be evaluated within the script itself – since the output format is key-value separated by `=`, the outcome is having the keys set as variables
- Exits returning the invalid stanza error if any of the variable is not found
- Prints the variables

Note Please note how each single error message is redirected to stderr following the best practices – this is accomplished by the trailing `> & 2`.

Set the script as executable, then run it. For example:

```
./credentials.sh ldap
```

It prints the variable's values of the `ldap` stanza. Try also running it with a non-existent stanza:

```
./credentials.sh cache
```

As expected, the script prints the error message and exits.

When dealing with `INI`-formatted files, most of the time, it is more effective writing a Python script rather than a shell script, since Python provides native facilities for processing the `INI` format by simply loading the `configparser` library.

To see it in action, create the `credentials.py` script with the following contents:

```
#!/usr/bin/python3
import sys
import os
import configparser

CREDENTIALS_FILE="credentials.ini"

try:
    if not os.path.isfile(CREDENTIALS_FILE):
        raise FileNotFoundError
    if not os.access(CREDENTIALS_FILE, os.R_OK):
        raise PermissionError
    stanza=sys.argv[1]

    credentials = configparser.ConfigParser()
    credentials.read(CREDENTIALS_FILE)

    print(f"username: {credentials[stanza]['username']}")
    print(f"password: {credentials[stanza]['password']}")

except FileNotFoundError as e:
    print(f"{CREDENTIALS_FILE} file not found!",
            file=sys.stderr)
    sys.exit(1)
except PermissionError as e:
```

```
    print(f"Unable to read {CREDENTIALS_FILE} file - Check "
        "permissions!",
            file=sys.stderr)
    sys.exit(1)
except IndexError as e:
    print("You must specify the stanza containing the "
            "variables to load", file=sys.stderr)
    sys.exit(1)
except KeyError as e:
    print(f"stanza {e} not found!", file=sys.stderr)
    sys.exit(1)
```

Set the script as executable, then run it. For example:

```
./credentials.py database
```

The script behaves the same way as the previous shell script does, with the difference of not needing to run an external utility for processing the `credentials.ini` file.

From the code length perspective, BASH still looks winning, but from the tidiness and cleanliness perspective, Python starts to look engaging. For example, the try-except block grouping the exceptions to be handled and the specific exceptions caught enables it to handle every exception in one place, resulting in a more clean and tidy approach.

Comma-Separated Values (CSV)

Note This paragraph's sample scripts are supposed to be stored in the *csv* directory beneath the chapter's workspace directory: create it and change directory to it – `cd ~/devsecops-pillars/ch06; mkdir csv; cd csv`.

The last basic data format we see in this chapter is the comma-separated values (CSV). This format is used to create a table where columns are fields, whereas field values are stored in one or more rows. CSV, despite being the most basic format that can be used to create (very poorly performing) database tables, is broadly used when dealing with data migrations.

In its most basic format, the field delimiter is the comma.

As an example, create the `data.csv` with the following contents:

```
ID,Aging,NameAndNick,Surname,Team
10,215,Marco Antonio,Carcano,Devops
11,1128,John,Doe,Devops
```

This file's format and contents are so basic to be easily processed even by a shell script. For example, create the `load.sh` BASH script with the following contents:

```
#!/bin/bash

CSV_FILE="data.csv"


function print_data() {
    echo "--------------------------"
    for ((i=0; i<${#HEADER[@]}; i++))
    do
        echo "${HEADER[i]}=${ROW[i]}"
    done
}

if [[ ! -f "${CSV_FILE}" ]]; then
    echo "${CSV_FILE} file not found!" >&2
    exit 1
fi
if [[ ! -r "${CSV_FILE}" ]]; then
    echo "Unable to read ${CSV_FILE} file" >&2
    exit 1
fi

CNT=0
while IFS= read -r LINE
do
    IFS=',' read -r -a ROW <<< "${LINE}"
    if [[ ${CNT} -eq 0 ]]; then
        HEADER=("${ROW[@]}")
    else
```

```
        if [[ -n "${LINE}" ]]; then
           print_data
        fi
    fi
    CNT=$(( CNT + 1 ))
done < ${CSV_FILE}
```

The script is very basic: after a few checks to ensure being able to access the contents of the `data.csv` file, it just reads it one line at time, storing it in the `LINE` variable. If the line is the first one, its contents are copied into the `HEADER` array, whereas all the other lines' contents are copied to the `ROW` array. These arrays, when they are not empty, are then processed inside the print function that displays each field's contents.

Note Please note how it sets the *IFS* variable to a comma: in *BASH*, the Internal Field Separator (*IFS*) is the variable containing the character used as the separator to tokenize contents. Note also the use of the here-string <<< to pass the text to the *read* command.

Set the script as executable, then run it:

```
./load.sh
```

Although the script works, mind this is a very basic scenario: both this CSV and the BASH script are quite fragile and have shortcomings indeed. The most straightforward shortcoming is not being able to handle numbers using commas as values. For example, storing `21,5` in place of `215` is impossible using such a simple implementation of the CSV format.

A quick and dirty fix can be using semicolons in place of commas as the values separator. In this case, the previous example becomes

```
ID;Aging;NameAndNick;Surname;Team
10;21,5;Marco Antonio;Carcano;Devops
11;1128;John;Doe;Devops
```

As for the `BASH` script, it is necessary to modify the `IFS` variable to use the semicolon in place of the comma. Anyway, this is just a workaround far from solving the real problem of having the field delimiter inside a field value. For example, a field with value `Marco;Antonio` breaks the standard again.

Note Another CSV variation is to use tab as the field delimiter. Mind that the CSV specifications are described in the IETF RFC 4180 and updated by RFC 7111, but they are both marked as memo.

A good way to mitigate the problem (but sadly it still does not completely sort it out) is wrapping field values between single quotes or double quotes. For example:

```
"ID","Aging","NameAndNick","Surname","Team"
"10","21,5","Marco Antonio","Carcano","Devops"
"11","11,28","John","Doe","Devops"
```

Or:

```
'ID','Aging','NameAndNick','Surname','Team'
'10','21,5','Marco Antonio','Carcano','Devops'
'11','11,28','John','Doe','Devops'
```

The problem now is that the complexity for processing the format is growing far beyond the capabilities of shell scripts and related command-line utilities, so requiring to switch to more powerful scripting languages, such as Python, that provides native facilities for processing it.

Note You can of course use system tools such as *grep*, *sed*, and *awk* to process *CSV* in shell scripts – several legacy scripts leverage on them, but always mind the performance tradeoff of spawning subshells every time they are run. In addition to that, an excessive use of these tools in scripts makes them harder to maintain because of the several tool-specific syntaxes to use, escaping, multiline statements handling, and so on. I'm not saying to avoid these tools – in this book, we have an entire section with some chapters dedicated to them. I'm just pointing out to each time you ask yourself which is the best option (shell script or scripting language) for your specific use case, minding criteria such as the delivery time, the maintainability, and ease (how easy it would be to improve and bug-fix it).

Let's go on and improve the previous example; create the `scientists.csv` file with the following more realistic contents:

```
"ID","Average","NameAndNick","Surname","Scientific Branch"
"1","1,5","Alfredo Erberto, ""AED""," Dupasquier","Physicist"
"2","1,28","Erwin ""ErS""","Schrödinger","Physicist"
```

As you can see, this time we have several challenges:

- **Use fields containing Umlaut**: Encoding point of attention
- **Put the , separator inside values**: Formatting challenge
- **Some of the fields contain "**: Formatting challenge, since it is used to enclose the fields

---

Note When having quotes inside the field values, remember to escape them by doubling – in this example, *"AED"* is quoted as *""AED""*.

Let's now create the `load.py` Python script with the following contents:

```
#!/usr/bin/python3

import sys
import csv

DATA_FILE="scientists.csv"

try:
    with open(DATA_FILE, newline="" , mode="r",
                encoding="utf-8") as table_fp:
        table = csv.DictReader(table_fp, delimiter=",",
                quotechar='"')
        for row in table:
            print(row)
except FileNotFoundError as e:
    print(f"{DATA_FILE} file not found!",
            file=sys.stderr)
    sys.exit(1)
```

```
except PermissionError as e:
    print(f"Unable to read {DATA_FILE} file - Check "
            "permissions!", file=sys.stderr)
    sys.exit(1)
```

Set the script as executable, then run it.:

```
./load.py
```

The script opens the file using the `open` statement and reads the `CSV` using the `csv.DictReader` method, passing as argument the delimiter and the quoting character in use in the CSV file. In addition to that, it also contains the exception handling code necessary to manage the access-related problems with the `scientists.csv` file that may arise.

It seems here Python starts shining over shell scripting: besides the cleaner and tidier style, this time also the lines of code are less. I'm of course not considering performances – these must be checked use case by use case.

Note Please note also the use of the `encoding` parameter of the `open` function: it is a very important parameter since it is used to set the character set encoding to match the one used when the file was created. An encoding mismatch results into improper characters rendering – if you replace `utf-8` with `iso8859_2`, you are turning Schrödinger into SchrĂśdinger.

Summary

In this chapter, we explored key-value, INI, and CSV data formats, practicing with examples in BASH and Python. These formats are commonly used for application configuration, with CSV files being particularly prevalent for data interchange in data pipelines.

This chapter has equipped you with the skills to write and manage automation scripts and integrations that require basic yet essential configurations. Understanding these data formats is crucial for DevOps and DevSecOps practitioners, as they enable efficient data handling, facilitate integration between tools and systems, and support automation processes. This knowledge ultimately contributes to more robust and secure software development practices within the DevOps framework.

PART IV

Data Processing

DevOps and DevSecOps professionals are not just supposed to know how to program using development tools: since they are required to facilitate processes, it is very common to write short but effective shell scripts or one-liners to quickly address integration problems.

The chapters in this section explore regular expressions (RegEx) and introduce three classic and high-performance command-line utilities: `grep`, `sed`, and `awk`.

Despite their age, these tools remain highly effective in specific scenarios due to their flexibility and performance.

By using them, it is often possible to achieve common data processing tasks by writing an efficient one-liner, which can save time compared to implementing the same logic in more sophisticated scripting languages like Python, while sometimes yielding better performance.

Moreover, it is common to encounter these utilities in legacy scripts that engineers are tasked with maintaining, troubleshooting, and improving as necessary.

Note When processing structured formats, it is often necessary to use format-specific matching clauses sometimes spanning multiple lines: although some use cases can be sorted out with also these tools, writing the proper matching expression becomes challenging and error-prone, so in such a situation it is better to use format-specific tools such as `xpath`, `jq`, and `yq`. We'll talk about them in the next section.

CHAPTER 7

Regular Expressions (RegEx)

Regular expressions (RegEx, for short) are a formal language used to define patterns for searching, matching, or substituting text that meets specific criteria.

These expressions provide powerful tools for text parsing and manipulation, which are essential for automating tasks and processing data in scripts: command-line utilities like `grep` and `sed` lose much of their power if users lack a solid understanding of regular expressions.

Another field where RegEx skills are mandatory is observability: here the use of capture groups and backreferences is particularly valuable for extracting operational and security information from log files, facilitating effective monitoring and analysis. This skill set enhances the ability to identify anomalies, ensure compliance, and contribute to a more secure and efficient environment.

RegEx is a vast topic – entire books are dedicated to it. This chapter serves as a quick guide, aiming to give readers the essentials needed to understand regular expressions and handle common daily use cases: after reading it, you will have a good understanding of basic regular expressions (BRE), extended regular expressions (ERE), and concepts from Perl-Compatible Regular Expressions (PCRE), which will enable you to efficiently configure pattern matching and extract relevant information from various data sources.

The Purpose of Regular Expressions

Regular expressions provide a way to express a pattern by using a standard syntax and check if a given text matches it: they are available in a lot of flavors (POSIX, PCRE, Python, JavaScript, Java, …) with their own nuances.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_7

Since POSIX is the most broadly used in Linux shell scripting, in this chapter we focus on this specific flavor, giving anyway also a few hints about PCRE since it has some interesting nuances.

Note PCRE stands for Perl-Compatible Regular Expression – if you need documentation on PCRE, you can install the `pcre-devel` RPM package. After this, you can view the PCRE syntax man by typing `man pcresyntax`.

In this chapter, we use the `egrep` command-line utility to provide some practical examples. Mind anyway that `grep` will be better described in the next paragraph.

Note POSIX regular expressions have been improved over the years: the original standard is now called basic regular expression (BRE), whereas the one with the new extensions is called extended regular expression (ERE). Although they share some basis, they have also differences that make them not compatible: for example, while in BRE the | character is considered literal and it must be escaped using a backslash to have it evaluated as special (it is the logical OR), in ERE it is just the opposite!

Meta-Characters

Meta-characters are reserved symbols used to assist in pattern matching. Table 7-1 summarizes the most used meta-characters.

Table 7-1. *RegEx meta-characters*

Character	Description
.	The dot is the wildcard character: this means that it can match any single character
^	The caret character identifies the beginning of the line – for example: `sudo grep "^Nov 10" /var/log/messages`
$	The dollar character represents the end of the line – for example `sudo grep "line ends like so$" /var/log/messages`

Here are some examples of using these meta-characters:

1) Searches for a line of only three characters in *foo.txt* file:
   `egrep "^...$" foo.txt`
2) Print the contents of a Python file stripping the lines starting with the # character.
   `egrep -v "^#" mycode.py`

Note In Python syntax, lines starting with the # character are commented-out lines.

If needed, it is of course possible to force the `.` character to be evaluated as literal by escaping it using a backslash. For example:

```
sudo egrep "192\.168\.50\.31" /var/log/secure
```

Class of Characters

It is possible to express a group (class) of characters by enclosing them between square brackets `[]` using a specific syntax, as depicted by the following examples:

1) A class of characters matching only `h` and `c` characters:
   `[hc]`
2) A class of characters matching any digit (`0` to `9`):
   `[0-9]`
3) A class of characters matching any alphas:
   `[a-zA-Z]`

For example, to get any line containing only numbers from the `foo.txt` file:

```
egrep "^[0-9]*$" foo.txt
```

Note The * character is a quantification meaning zero or more – we'll see quantification in detail very soon.

For our convenience, there are also pre-defined built-in character classes as summarized by Table 7-2: they must be listed beneath square brackets [ ] in the same way as the above ones.

Table 7-2. *RegEx built-in-character classes*

Character	Description
[:digit:]	Match any digits (0 to 9)
[:alnum:]	Match any alphanumeric character – from 0 to 9 OR from A to Z or from a to z
[:alpha:]	Match any alpha character from A to Z or from a to z
[:lower:]	Match any lowercase alpha character from a to z
[:upper:]	Match any uppercase alpha character from A to Z
[:blank:]	Match space and TAB characters only
[:punct:]	Match any punctuation characters [][!"#$%&'()*+,./:;<=>?@\^_`{|}~-]

The previous example, revised using the built-in character classes, becomes:

```
egrep "^[[:digit:]]*$" foo.txt
```

Shorthand

To ease readability, certain character classes can also be represented by shorthand as summarized by Table 7-3.

Table 7-3. *RegEx shorthand*

Character	Description
\d	Match any digit from 0 to 9
\D	Negates \d
\w	Match any word character from A to Z or from a to z, including digits and the characters and _
\W	Negates \w

(*continued*)

Table 1-2. (*continued*)

Character	Description
\s	Match any space character – this includes TAB, newline, and so on
\S	Negates \s
\b	Match a word boundary
\B	Negates \b

Quantification

The symbols in Table 7-4 are used to quantify the number of times the meta-character or class of characters should be repeated to have a match.

Table 7-4. *RegEx quantification*

Character	Description
*	The previous character or class matches zero or more times.
?	The previous character or class should match at most one time – this means that it can also match zero times.
+	The previous character or class matches at least one time.

For example, we can get the entries of the /var/log/messages file containing the search pattern ntpd[N] where N is any integer number by running

```
sudo egrep "ntpd\[[[:digit:]]+\]" /var/log/messages
```

This extracts every line written by the ntpd process no matter which is the process id.

Note BRE requires to escape both + and ? using a backslash to prevent them from being evaluated as literals.

The preceding example, rewritten as BRE, becomes

```
sudo grep "ntpd\[[[:digit:]]\+\]" /var/log/messages
```

About quantification, it is certainly worth mentioning that, conversely from POSIX *RegEx*, which implements only the *greedy* behavior, *PCRE* implements three different flavors of quantification: greedy, reluctant, and possessive.

Let's test them one by one checking how they affect the matching of the string containing "https://grimoire.carcano.ch/p1/p2/p3/foofile.html".

First, in a shell, set the URL variable as follows:

```
URL="https://grimoire.carcano.ch/p1/p2/p3/foofile.html"
```

Since grep by default implements the POSIX RegEx, to see them in action, we must enable PCRE support by passing the -P option: just run each statement in Table 7-5 and look at the outcome.

Table 7-5. *Perl RegEx quantification flavors*

char	Flavor	Behavior
*	**greedy**	The default – it enlarges the match as much as possible – for example: `echo "${URL}" | grep -P "https://.*/"` The highlighted part of the output is https://grimoire.carcano.ch/p1/p2/p3/
*?	**reluctant**	Sometimes also lazy – it considers the match as small as possible – for example: `echo ${URL} | grep -P "https://.*?/"` The highlighted part of the output is https://grimoire.carcano.ch/
*+	**possessive**	Like greedy, it enlarges the match as much as possible but, conversely, when it reaches the end, it does not backtrack looking for smaller matches. For example: `echo "${URL}" | grep -P "https://[^/]*/?([^\/]+\/)*+"` The highlighted part of the output is https://grimoire.carcano.ch/p1/p2/p3/ The `([^\/]+\/)` defines a capture group capturing one or more characters that are not slashes (i.e., a folder name) followed by a slash. The * quantifier claims the capture group can repeat 0 or more times, and the + makes the quantifier possessive.

Note Always mind that everything has a cost: in terms of resources, because of its intensive backtracking in case of failing matches, the reluctant quantifier "costs" much more than the greedy one … so use it in your scripts only when you really need it. The possessive quantifier may perform even better than the greedy one … but writing a possessive RegEx can be much more challenging than a greedy one.

For your convenience, Table 7-6 summarizes quantifiers and the flavor-specific syntax.

Table 7-6. *RegEx quantifiers and flavor summary*

Quantifier	POSIX RegEx		
		Perl RegEx	
	Greedy	Lazy	Possessive
Zero or more	`*`	`*?`	`*+`
One or more	`+`	`+?`	`++`
Zero or none	`?`	`??`	`?+`

It is of course possible to state the number of times the match should happen.

Table 7-7. *RegEx exact quantifiers*

Character	Description	Example
{n}	The previous character or class matches exactly *n* times.	`egrep "^[0-9]{5}$" foo.txt`
{n,}	The previous character or class matches *n* or more times.	`egrep "^[0-9]{1,}$" foo.txt`
{,m}	The previous character or class matches at most *m* times.	`egrep "^[0-9]{,5}$" foo.txt`
{n,m}	The previous character or class matches at least *n* times, but not more than *m* times.	`egrep "^[0-9]{1,5}$" foo.txt`

Note BRE requires to escape both { and } using a backslash to prevent them from being evaluated as literals. So, for instance, the first of the above examples would be `grep "^[0-9]\{5\}$" foo.txt`.

Negating a Class of Characters

When the ^ (*caret*) meta-character precedes a character class, it negates the class of characters to match. For example, a simple way for specifying consonants is to negate a class of character with only vowels:

```
[^aeiouAEIOU]
```

To see it in practice, we can write a statement to get the lines containing only consonants: in this case the class matching consonants `[^aeiouAEIOU]` must be quantified as matching at least one or more times using the + and `.` In addition to that, the match must start from the beginning of the line ^ to the end of the line $. So, the final RegEx is

```
^[^aeiouAEIOU]+$
```

We can test it with egrep as follows:

```
egrep  "^[^aeiouAEIOU]+$" /usr/share/dict/linux.words
```

Note You may need to install the `words` RPM package for the above example to work – `sudo dnf install -y words`.

Logical OR

Sometimes it is necessary to specify multiple matching patterns having even just a match on any of them as the necessary condition for a positive outcome, … in other words having multiple matching patterns bound by a logical OR. The character assigned to the logical OR is the pipe |, but ERE and BRE implement it in different ways: ERE evaluates | as logical OR, using it as literal only if escaped by a backslash. BRE works exactly the opposite way.

As an example, if you want to extract the entries of the root and daemon users from the `/etc/passwd` file

```
egrep "^daemon:|^root:" /etc/passwd
```

Capturing Group and Backreferences

A capturing group is used to mark a part of a RegEx to capture the string matching that specific marked part.

RegEx (especially PCRE2) and capture groups are mandatory skills to work with Security Information and Event Management (SIEM) software: the system relies on them to extract data from various data streams and put the extracted values into capture groups that are later processed by the SIEM rules.

To specify a capture group, the matching RegEx must be enclosed by parentheses `()`. For example, the RegEx

```
ID=([[:digit:]]+)
```

matches entries containing `ID=` followed by any integer number, such as

```
ID=123
```

capturing the value 123 in the capture group.

Note If necessary, you can add a quantifier right after the capture group, as well as – if working with PCRE – mark it as greedy, reluctant, or possessive. The example we made for the possessive quantifier works exactly like that.

It is of course possible to define more capturing groups: their respective captured strings are stored within an array so that the array's item number matches the capture group's number. This can be exploited for example to process the `~/devsecops-pillars/ch06/csv/scientists.csv` file we created in the previous chapter – the RegEx with the capture groups to capture each single field is

```
^"([[:digit:]]+)","(.*)","(.*)","(.*)","(.*)"
```

Backreferences let you retrieve and reuse the text captured by each group. For example, given the second row of the CSV (skipping the header line), the value captured by group 1 is `1`, by group 2 is `1,5`, by group 3 is `Alfredo Erberto, "AED"` (mind how it gets also `"AED",` since its double quotes are escaped inside the field), by group 4 is `'Dupasquier'` (mind how it gets also the space, since it is in the field within the double quotes), and by group 5 is `Physicist`. You can then refer to these values in your RegEx as `\1`, `\2`, `\3`, `\4`, and `\5`.

The following example shows how capture groups and backreferences enable us to extract specific parts of the row, for example, the name and surname:

```
tail -n +2 ~/devsecops-pillars/ch06/csv/scientists.csv \
| sed -E 's/^"([[:digit:]]+)","(.*)","(.*)","(.*)","(.*)"/
NAME=\3;SURNAME=\4/; s/""/"/g'
```

The `tail` statement prints the `~/devsecops-pillars/ch06/csv/scientists.csv` file skipping the first line, whereas the `sed` statement uses the substitute command (`s`) two times: the first one to define a capture group dedicated to each single field, substituting the text with `NAME=` followed by the value of the third capture group and `SURNAME=` followed by the value of the fourth capture group; the second time it replaces `""` with `"`.

Note Don't worry if the above sed statement is still not very clear to you: we'll discuss `sed` in detail later in a dedicated chapter of this section.

Capture groups and backreference are supported also by the `grep` command-line utility: as an exercise, let's assume you want to extract every line of the `/etc/passwd` file having a `uidnumber` the same as the `gidnumber`. The statement to run is

```
grep -P '^\w*:x:([[:digit:]]+):\1:' /etc/passwd
```

This pattern matches every line starting with a word (the username, in this case) followed by a colon, then a `x` character followed by another colon, then a digit (the `uidnumber`). This is the spot where the `([[:digit:]]+)` capture group captures the number. Then there is another colon followed by another number (the `gidnumber`) – here is where the backreference becomes effective for filtering the lines, requiring the `gidnumber` to match what we captured.

Note Mind that `grep` implements backreference only when running in PCRE mode – that's why we are running it with the `-P` option.

Summary

In this chapter, we provided an overview of regular expressions (RegEx), covering basic regular expressions (BRE), extended regular expressions (ERE), and concepts from Perl-Compatible Regular Expressions (PCRE). We also learned how to extract information from text efficiently using capture groups and backreferences.

For DevOps and DevSecOps professionals, mastering regular expressions is crucial for automating tasks such as data extraction, input validation, and error detection, significantly enhancing productivity. Understanding the nuances of different RegEx flavors (BRE, ERE, and PCRE) allows for greater flexibility in scripting and tool integration, enabling adaptation to diverse environments and requirements.

Ultimately, proficiency in RegEx empowers practitioners to maintain high standards of quality, security, and efficiency in their workflows.

CHAPTER 8

Working with Grep

In this chapter, we will explore one of the most famous and widely used Unix tools: `grep`.

This utility, which excels in data validation, automation, log analysis, and security monitoring, can scan single files and entire directory trees for data matching specific lookup criteria, such as exact strings or regular expressions (RegEx).

Its flexibility makes it valuable both as a command-line utility for operational tasks and as a convenient, small-footprint component in shell scripts.

Proficiency in `grep` allows professionals to efficiently search through large volumes of data, quickly identify patterns, and extract relevant information, which is crucial for troubleshooting and ensuring system integrity.

This capability enhances operational security by enabling timely detection of anomalies and potential threats within logs and configurations.

As a fundamental skill in the DevOps toolkit, mastery of `grep` significantly contributes to improved efficiency and effectiveness in managing and securing applications and infrastructure.

Note Before going any further, create this **chapter's workspace directory** by running `mkdir -p ~/devsecops-pillars/ch08`, and then immediately navigate to it by running `cd ~/devsecops-pillars/ch08`.

What Is Grep

Grep – an acronym for Global Regular Expressions Print – is an evergreen command-line tool for searching lines matching regular expressions and printing them to the standard output.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_8

Written by Ken Thompson himself, it was later improved over the years, evolving it into `egrep` (to provide support to the extended regular expressions) and `fgrep` (searching using the Aho–Corasick string matching algorithm – this topic is out of scope for this book).

By default, `grep` processes basic regular expressions, but it can be configured to use extended regular expressions, by passing the `-E` option, or even PCRE (Perl-Compatible Regular Expressions), by using the `-P` option.

The tool is invaluable not only while performing operating tasks, such as filtering log files looking for lines matching specific criteria; it is also one of the most used bricks in the shell scripts' pipelines.

Scanning Files

The most trivial usage is looking for lines matching a pattern into a file – for example:

```
sudo grep " kernel: " /var/log/messages
```

prints every line containing the word `kernel` followed by a colon, wrapped by a space character – the use case is looking for kernel messages.

Note In this section, since access to `/var/log` requires administrative rights, every grep statement related to the `/var/log` path is prepended by `sudo`, which grants admin rights to the statement. We will discuss the `sudo` command-line utility in detail in Chapter 21.

Grep by default does not color the values matching the pattern – this behavior can be enabled by passing the `--color` command-line option. Anyway, mind that in modern Linux distributions `grep` is an alias to the real `grep` command-line utility – you can easily see this by typing

```
alias grep
```

The output is

```
alias grep='grep --color=auto'
```

So, you can just run `grep` and get the matching text colored.

Note If you are using Oracle Linux, the `root` user does not have an alias set for the `grep` command: as a result, also when you run `grep` wrapped by `sudo`, unless you explicitly specify `--color=auto`, you will not get colored text.

Scanning All Files in a Directory Tree

It is possible to have grep scanning every file in a directory tree by specifying the `-R` command-line option.

For example:

```
sudo grep -R " kernel: " /var/log
```

In this case the output shows the path to the file containing the matching line.

Case-Insensitive Search

If instead you need to perform a case-insensitive scan, just specify the `-i` command-line option.

For example:

```
sudo grep -i " kErNEl: " /var/log/messages
```

Inverting the Matching Criteria

If necessary, it is possible to invert the matching criteria by specifying the `-v` command-line option.

For example, to show every line not containing the expression `" kernel: "`

```
sudo grep -v " kernel: " /var/log/messages
```

Note You can combine multiple `grep` options as necessary. For example, `sudo grep -Ri " kErNEl: " /var/log` performs a case-insensitive search in the whole `/var/log` directory tree.

Piping to Grep

Although its original purpose was working with regular files, grep also supports input from pipe – for example, to filter only the lines of output of the ps command-line utility containing the sshd word, type

```
ps aux | grep "sshd"
```

The outcome is the list of running sshd processes, plus the grep process itself. The line with the grep process is matching because the statement contains the sshd word: if you are really looking for only the sshd processes, a smarter statement would be

```
ps aux | grep "[s]shd"
```

The [s]shd RegEx indeed matches the lines with the word sshd, but not the line with the grep statements, since it contains the [s]shd word. The outcome is having the line with the grep process itself excluded from the returned output.

Validating Standard Input

A very common need accomplished using grep is validating the standard input – for example, launch this statement in your shell:

```
egrep '\b((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\.|$)){4}\b'
```

then, start typing IPv4 addresses followed by pressing enter, …. valid entries (the range is 0.0.0.0–255.255.255.255) are printed in a new line colored in red.

When you are done with the trial, just press CTRL+C to quit.

Note In this example we used egrep because the above one is an extended regular expression, but the same outcome can be achieved by running grep with the -E option.

Using Grep in Shell Scripts

As we said, grep is often used as a brick inside shell scripts. In this example we reuse the same extended regular expression to create the validate-ip.sh script with the following contents:

```
#!/bin/bash
if ! echo "${1}" | grep -Eq '\b((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\.|$)){4}\b'; then
  echo "Invalid IP ${1}" >&2
  exit 1
fi
```

The script, within the if statement, inside a subshell, prints the first argument passed to it using echo and pipes the output to the grep statement: because of the -q option, the grep does not output anything – it just exits returning 0 if the pattern matches or 1 if it does not match.

In addition to that, while using grep, we also enable extended regular expression support by using the -E option. The exclamation mark right after the if negates the outcome, so the block is entered only when the pattern does not match.

To test it, assign it execution rights and run it as follows:

```
./validate-ip.sh 255.255.255.255
```

Since this is a valid IP address, the script does not return output. Re-run it as follows:

```
./validate-ip.sh 255.255.255.256
```

This time it fails as follows:

```
Invalid IP 255.255.255.256
```

As expected, it also exists with a return code different from 0: you can check it by typing echo $? in the shell prompt.

Summary

In this brief chapter, we provided a quick example-driven walkthrough of working with `grep`.

This versatile tool offers numerous applications for addressing common challenges in data validation, automation, log analysis, and security monitoring.

In a fast-paced development environment, the ability to quickly filter through large volumes of log files and configuration scripts allows professionals to identify issues, track changes, and ensure compliance with security policies. This enables searching for specific patterns, making it easier to pinpoint errors or anomalies that could affect application performance or security. Additionally, integrating `grep` into automated scripts enhances efficiency, allowing for real-time analysis and reporting.

Mastery of this tool not only streamlines workflows but also contributes to a more proactive approach in maintaining system integrity and operational excellence.

As a result, proficiency in `grep` is an essential skill for DevOps and DevSecOps professionals.

CHAPTER 9

Working with Sed

The Stream EDitor (`sed`) is a very effective stream manipulation command-line tool allowing for powerful text processing capabilities. Despite being quite an old tool, it is still nowadays one of the preferred and most effective tools to accomplish quite common data processing tasks: its concise syntax indeed enables writing effective one-liners in a far small portion of the amount of time that would be required to write the same logic with a scripting language such as Python.

For DevOps and DevSecOps professionals, proficiency in `sed` can save time by eliminating the need to create unnecessary scripts for tasks that can be efficiently handled with a concise one-liner, which can also be easily embedded in shell scripts as necessary. Additionally, it is common to encounter `sed` statements in legacy scripts that engineers are responsible for maintaining.

Understanding sed's line-oriented design, including the use of pattern space and hold space, enables professionals to perform precise modifications while minimizing the risk of accidental changes.

Mastery of sed's features enhances the ability to automate tasks and integrate text processing into various workflows, significantly improving the efficiency of DevOps and DevSecOps practices.

As a relevant tool in modern environments, proficiency in sed empowers professionals to handle data transformations effectively, contributing to streamlined operations and better management of configuration files and logs.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_9

Note `sed` is not the only utility that can perform this kind of transformations: there are other tools, more specialized, such as `tr`, or more general purpose, such as `awk`, which instead is a pattern scanning and processing language. When dealing with structured formats, such as XML, JSON, and YAML, it is instead best to use format-specific tools when available. For example, if you are dealing with XML, use `xmlstarlet` instead. We will discuss it in Chapter 11, when talking about XML.

In this chapter we delve into using `sed` to write one-liners that can be embedded in shell scripts or used to address common system administration tasks effectively.

Note Before going any further, create this **chapter's workspace directory** by running `mkdir -p ~/devsecops-pillars/ch09`, and then immediately navigate to it by running `cd ~/devsecops-pillars/ch09`.

An Initial Walkthrough

Sed is really a hard nut to crack: before delving into its syntax, it is better to have a quick walkthrough to explain its usage with a few examples.

Invocation

The basic invocation of `sed` is as follows:

```
sed 'expressions;list' opt_file_to_proc_1 opt_file_to_proc_2
```

This statement instructs `sed` to load the data to be processed from `opt_file_to_proc_1` and `opt_file_to_proc_2` files, processing them using the `'expressions;list'` list of statements.

Note It is possible to specify as many files as necessary.

An alternate way of invoking sed is by piping data to it. For example:

```
echo 'a stream to process' | sed 'list;of;expressions'
```

Note Be wary that sed by default uses basic regular expressions: if you want to use extended regular expression, then you must specify the -r command option.

---

sed can load the expressions to be used to process the input stream either

- From an expressions file
- By reading expressions specified as one-liner

As an example, consider the following two expressions:

```
s/one/1/g
s/two/2/g
```

We can process them using a one-liner like the following:

```
echo "one two three one four" | sed 's/one/1/g;s/two/2/g'
```

The outcome is as follows:

```
1 2 three 1 four
```

An alternate way is to provide each expression as a dedicated argument repeating the -e command option multiple times:

```
echo "one two three one four" |sed -e 's/one/1/g' -e's/two/2/g'
```

To see an example of how to invoke sed using an expressions file, create the foo.sed expressions file with the following contents:

```
# an example expression file
s/one/1/g
s/two/2/g
```

then run sed using -f command option to specify the expressions file to be processed:

```
echo "one two three one four" | sed -f foo.sed
```

Note Although lines beginning by # are generally treated as comments, mind there are some versions or specific contexts in which this convention may not hold true, and lines starting with # may not be interpreted as comments.

If necessary, just as you can do with `-e` option, you can specify multiple expressions files by providing them one by one repeating the `-f` option.

It is also possible to combine expression files and one-liners within the same invocation by using the `-e` command option to provide additional expressions:

```
echo "one two three one four" | sed -f foo.sed \
-e 's/three/3/' -f bar.sed
```

In-place File Modification

The most common use case with `sed` is modifying streams sourced from files.

Note Since `sed` by default prints to the standard output the modified stream, you might be tempted to think that the way for writing changes to a file is redirecting the output of `sed` using the > operator, such as in the statement `sed -e 's/one/1/g' -e 's/two/2/g' file_to_process.txt > file_to_process.txt`. This approach is wrong: as discussed in Chapter 5, when the shell processes output redirections, it opens the target file for writing, which truncates the existing contents before `sed` has a chance to read from it. Consequently, `sed` ends up processing a truncated file, leading to the loss of the original data. Therefore, using this method will result in wiping the contents of your source file.

As a real-life example of an in-place file modification, let's see how to configure `chronyd` to use the servers of the Swiss NTP pool.

In this example, we replace the current NTP pool setting in the `/etc/chrony.conf`.

First, let's make a backup copy of the original file:

sudo cp -p /etc/chrony.conf /etc/chrony.conf.bak

To write the sed expression we must

- Guess a matching regular expression to select the line to change
- Guess the command to be used

A RegEx that matches the line which starts with the word pool is ^pool.
We must then focus on writing the sed command that

- Specifies the action to perform – in this case we need to change (replace) the whole line, so we specify the c command
- Provides the argument of the command; that in this case is the new text used to replace the old one

The whole sed change command, along with its argument, is as follows (mind it is all in one line):

```
c server 0.ch.pool.ntp.org\nserver 1.ch.pool.ntp.org\nserver 2.ch.pool.ntp.
org\nserver 3.ch.pool.ntp.org
```

Since the new text must span multiple lines, we marked the end of each line using the line feed control character \n.

The whole sed expression is (mind it is all in one line)

```
/^pool /c \server 0.ch.pool.ntp.org\nserver 1.ch.pool.ntp.org\nserver 2.ch.
pool.ntp.org\nserver 3.ch.pool.ntp.org
```

We can run it as a one-liner as follows (mind it is all in one line):

```
sed '/^pool /c \server 0.ch.pool.ntp.org\nserver 1.ch.pool.ntp.org\nserver
2.ch.pool.ntp.org\nserver 3.ch.pool.ntp.org' /etc/chrony.conf
```

Once run, it just prints the modified stream to the standard output, but the /etc/chrony.conf file remains unchanged: this is expected, since as we said this is the default behavior of sed.

To have sed modifying the input file, we must supply the -i option (in-place modification):

```
sudo sed -i '/^pool /c \server 0.ch.pool.ntp.org\nserver 1.ch.pool.ntp.org\
nserver 2.ch.pool.ntp.org\nserver 3.ch.pool.ntp.org' /etc/chrony.conf
```

This time the command does not print anything to the standard output and just performs the file change.

In this specific example, since modifying `/etc/chrony.conf` requires administrative rights, we are using `sudo` to run it as the `root` user.

Note If you were really configuring `chrony` with the above `sed` one-liner, you must then restart the `chronyd` systemd unit to have it load the new configuration – the command to run is `sudo systemctl restart chronyd`.

Let's restore the `/etc/chrony.conf` file as it was before:

```
sudo mv  /etc/chrony.conf.bak /etc/chrony.conf
```

For the sake of completeness, mind that the preferred way to create backup files is providing a suffix, such as `.bak`, right after the `-i` command-line parameter: this tells sed to create a backup file with the same name as the original one ending by the specified suffix. For example:

```
sudo sed -i.bak '/^pool /c \server 0.ch.pool.ntp.org\nserver 1.ch.pool.ntp.
org\nserver 2.ch.pool.ntp.org\nserver 3.ch.pool.ntp.org' /etc/chrony.conf
```

One of the nice side effects of backup file is that they provide a handy way for checking the changes by comparing the differences:

```
diff /etc/chrony.conf /etc/chrony.conf.bak
```

Before going on, restore the `/etc/chrony.conf` file as it was originally:

```
sudo mv /etc/chrony.conf.bak /etc/chrony.conf
```

Record Matching Clauses

Some use cases require limiting the changes to only those lines matching a specific criterion. Let's illustrate it with an example.

Let's suppose the `/etc/fstab` file contains the following entry:

```
LABEL=tmp /tmp  xfs defaults,noatime 0 0
```

Now suppose we want to add the mount options `noexec,nodev,nosuid` to the `/tmp` mount point, preserving the existing ones.

In this case, we can use `defaults` as the pattern to be matched by the `s` command to perform the change. But this time it is more challenging than before: multiple lines indeed contain the `defaults` word, so we must limit the change only to lines containing the " `/tmp` " pattern (mind the spaces).

This is achieved by specifying the record matching criteria for selecting only the proper line: this line-matching expression must precede the `s` command and be enclosed between slash characters.

The final expression looks as follows:

```
/ \/tmp / s/defaults/defaults,noexec,nodev,nosuid/
```

where

- `/ \/tmp /` provides the record matching clause, limiting the expression to affect only lines containing " `/tmp` " (mind the spaces in the clause)
- `s/defaults/defaults,noexec,nodev,nosuid/` tells sed to run the substitutes (`s`) command for replacing the text matching the pattern `defaults` with `defaults,noexec,nodev,nosuid`

After combining the record matching criteria and the expression to perform the change, the one-liner to run is

```
sed \
'/ \/tmp / s/defaults/defaults,noexec,nodev,nosuid/' \
/etc/fstab
```

Showing Only Changes

Since the purpose of `sed` is to modify a stream and print the entire modified output to the standard output, it is expected behavior for it to display the whole modified stream. However, when experimenting to find the correct expression for your use cases, this behavior can be confusing and frustrating, as you are constantly overwhelmed by the whole stream, needing to scroll through the output to locate the specific changes and verify that the sed command worked as intended.

While experimenting, it is possible to use a combination of

- The `-n` sed command-line option, which disables printing the stream to the standard output
- The `p` sed expression command to print the currently changed record to the standard output

Let's see this trick in action applying it to the last one-liner we used:

```
sed -n \
'/ \/tmp / s/defaults/defaults,noexec,nodev,nosuid/p' \
/etc/fstab
```

As you can see, we specified the `-n` sed option, which disables printing to the standard output, and, at the end of the `sed` expression, we added the `p` command to print the applied change.

The generated output is just

```
LABEL=tmp /tmp xfs defaults,noexec,nodev,nosuid,noatime 0 0
```

So, as requested, it printed only the changed lines, hiding all the other ones.

The Pattern Space

Sed is a line-oriented tool: it reads and processes the stream line by line.

Note Mind the consequence of being line oriented is that you cannot use a regular expression to match text which spans multiple lines.

Internally, while loading each line, `sed` removes the trailing newline characters and places the contents into a buffer known as the `pattern space`. This buffer

- Gets overridden with each read cycle.
- Serves as the source of the stream processed by the commands.
- Is also used to store the outcome of the commands, which replaces its current contents. If there are other commands or expressions, they will use the result of the previous command or expression as the source stream.

You can control how sed loads the pattern space using the commands in Table 9-1.

Table 9-1. *Sed pattern space commands*

Character	Description
n	Read the next line from the stream and put it into the pattern space (overwriting the current contents)
N	Read the next line from the stream and append it to the pattern space

For instance, if you have a record that spans multiple lines, you can use the N command in sed to load additional lines into the pattern space. This allows you to process multiple lines together, enabling you to work with records that are longer than a single line.

As an example, create the os.txt file with the following contents:

```
Id=1
Name=CentOS Linux
Version=8
Id=2
Name=CentOS Linux
Version=7
Id=3
Name=CentOS Linux
Version=6
```

Suppose we need to replace the value of the Name field with the string Rocky Linux only for the record with Version=8.

As you can see, from the keys perspective we have a recurring pattern (Id, Name, and Version) spanning three lines: this means that, to process this kind of content, we must process records of three lines in length.

Since the first line gets loaded already in the pattern space, to load also the remaining two lines to complete the record, we must also add the N command two other times. So, our expression must begin with

```
N;N
```

Since we want to limit the change to records containing `Version=8`, we must specify the following record matching criteria:

```
/Version=8/
```

Lastly, we must specify the actual sed command. In this case we use the `s` command to replace the `Name` field's value with the string `Rocky`:

```
s/Name=[^ \n]*/Name=Rocky/
```

Let's put everything together: this is the whole `sed` one-liner:

```
sed 'N;N;/Version=8/ s/Name=CentOS Linux/Name=Rocky/' os.txt
```

The output is

```
Id=1
Name=Rocky Linux
Version=8
Id=2
Name=CentOS Linux
Version=7
Id=3
Name=CentOS Linux
Version=6
```

The pattern space can also be used to deal with matching criteria spanning more than one line. As an example, modify the contents of the `os.txt` file to look as follows:

```
Id=1,Name=CentOS
Linux,Version=8
Id=2,Name=CentOS
Linux,Version=7
Id=3,Name=CentOS
Linux,Version=6
```

Here, because of the end of line between `CentOS` and `Linux`, the matching pattern to use for the replace command is `Name=CentOS \nLinux`. Anyway, since sed is line oriented, if you run the sed expression without including the `N` action to read other lines into the pattern space, the pattern will never match.

For example, the following expression won't work:

```
sed '/Version=8/ s/Name=CentOS[ ]*\n*[ ]*Linux/Name=Rocky \nLinux/' os.txt
```

whereas this one works as expected:

```
sed 'N;/Version=8/ s/Name=CentOS[ ]*\n*[ ]*Linux/Name=Rocky \
nLinux/' os.txt
```

The golden rule when playing with the pattern space is that it works as long as the record's length in lines remains the same. For example, if you rework the `os.txt` file to contain

```
Id=1,Name=CentOS Linux,Version=9
Id=2,Name=CentOS
Linux,Version=8
Id=3,Name=CentOS Linux,Version=7
Id=4,Name=CentOS Linux,Version=6
```

the run of the sed statement

```
sed 'N;/Version=8/ s/Name=CentOS[ ]*\n*[ ]*Linux/Name=Rocky \
nLinux/' os.txt
```

produces the following result:

```
Id=1,Name=CentOS Linux,Version=9
Id=2,Name=CentOS
Linux,Version=8
Id=3,Name=Rocky
Linux,Version=7
Id=4,Name=CentOS Linux,Version=6
```

revealing the change is now applied to the wrong record. This is caused by the misalignment of the lines read into the pattern space and the matching criteria used by the replace command.

Hold Space

Sed lets you use an additional buffer called `hold space`: conversely from the `pattern space` that gets overwritten by the next line of the stream at each read, the `hold space` persists until the end.

Its purpose is to provide a buffer where you can store data you can recall when needed.

Sed provides a few commands to manage the hold space as shown in Table 9-2.

Table 9-2. *Sed hold space commands*

Character	Description
h	Copy the pattern space contents to the hold space (overwriting the current contents)
H	Append the pattern space contents to the hold space
g	Copy the hold space contents to the pattern space (overwriting the current contents)
G	Append the hold space contents to the pattern space

To illustrate it, we use a purely didactical example that addresses the problem we are currently having in processing the `os.txt` file caused by having records with different length in lines:

```
Id=1,Name=CentOS Linux,Version=9
Id=2,Name=CentOS
Linux,Version=8
Id=3,Name=CentOS Linux,Version=7
Id=4,Name=CentOS Linux,Version=6
```

This time, we process the following one-liner:

```
sed -n 'H;$g;s/CentOS[ ]*\n*[ ]*Linux,Version=8/Rocky \nLinux,Version=8/
g;$p' os.txt
```

The output is

```
Id=1,Name=CentOS Linux,Version=9
Id=2,Name=Rocky
Linux,Version=8
Id=3,Name=CentOS Linux,Version=7
Id=4,Name=CentOS Linux,Version=6
```

This expression worked as expected. Let's have a close look at it:

- At each read, it appends the read line to the hold space. Since the next action is prepended by a `$`, sed keeps on reading the next line and adding it to the hold space until the end of the file is reached.
- It replaces the content of the pattern space with the contents of the hold space using the `g` command.
- It performs the substitution using the `s` command – since we have the whole file in the pattern space, we can match the regular expression across more than just one line.
- Only at the end (note the use of the `$`), it prints the contents of the pattern space using the `p` command.

Note Be careful with the amount of data you send to the hold space – this is an extreme didactic-only example where we load everything in the hold buffer, but usually you can do more tailored things.

Please mind we used this example just to show you sed internals: it is indeed possible to do the same (loading the whole file contents in the pattern space) simply by using the `-z` option as follows:

```
sed -z 's/CentOS[ ]*\n*[ ]*Linux,Version=8/Rocky \nLinux,Version=8/' os.txt
```

Note While making trials with copying/appending data into hold space and pattern space, you may benefit using the `l` command: it lists the current line in a visually unambiguous form.

Sed Expression Syntax

After practicing with some examples, we can now delve into sed starting from learning the basic syntax of a sed expression:

```
/LINE_MATCHING_CLAUSE/ statement
```

As you can see from above, an expression is made of

- `LINE_MATCHING_CLAUSE`: This is an optional filter that specifies the pattern to match. If the `LINE_MATCHING_CLAUSE` matches the contents currently loaded into the pattern space, the associated statement will be executed.
- `statement`: This is the action to be performed if the `LINE_MATCHING_CLAUSE` matches.

Note If the `LINE_MATCHING_CLAUSE` is omitted, the match always succeeds, causing the statement to be executed for every line.

A `sed` statement can either be

- **A `command`**: This includes the command itself along with any necessary options. You can also specify a list of commands to be executed sequentially. The syntax of a command is `/LINE_MATCHING_CLAUSE/ command/OPTIONS/`
- **Another sed expression**: This means that the statement can be another expression preceded by a `LINE_MATCHING_CLAUSE`. The syntax for this is `/LINE_MATCHING_CLAUSE/ expression`.

When the statement is another expression, things can become more complex, as the expression can include various commands and options. For more details on handling nested expressions, refer to the section on "Nesting Expressions."

Nesting Commands

It is possible to nest a list of commands within the same `LINE_MATCHING_CLAUSE` by

- Enclosing the list by curly braces {}
- Separating list items using semicolons (`;`)

The syntax is as follows:

```
/LINE_MATCHING_CLAUSE/ {command/OPTIONS/;command/OPTIONS/}
```

Mind that the nested commands run in the order they are listed.

To see a working example of nested commands within the same LINE_MATCHING_CLAUSE, since we are operating in a lab, we create a fictional NFS exports configuration, and then we modify it using sed.

Create the /etc/exports.d/devsecopsbook.exports NFS exports file with the following contents:

```
/srv/nfs/foo 192.168.0.0/255.255.255.0(rw,root_squash,sec=krb5:krb5i:krb5p)
/srv/nfs/bar 192.168.0.0/255.255.255.0(rw,root_squash,sec=krb5:krb5i:krb5p)
/srv/nfs/baz 192.168.0.0/255.255.255.0(rw,root_squash,sec=krb5:krb5i:krb5p)
```

Now let's assume that we need to alter only to line with the /srv/nfs/bar share as follows:

- Add the no_all_squash NFS export option.
- Modify the subnet mask of the subnet that is granted access to the share to 255.255.240.0.

The whole sed one-liner is

```
sed -i '/^\/srv\/nfs\/bar / {s/\(rw[^)]\)/\1no_all_squash,/;s/255\
.255\.255\.0/255.255.240.0/}' /etc/exports.d/devsecopsbook.exports
```

where:

- the LINE_MATCHING_CLAUSE is: "^\/srv\/nfs\/bar "
- the 1st command is: "s/\(rw[^)]\)/\1no_all_squash,/"
- the 2nd command is: "s/255\.255\.255\.0/255.255.240.0/"

After running it, the changed line in the /etc/exports.d/devsecopsbook.exports file must look as follows:

```
/srv/nfs/bar 192.168.0.0/255.255.240.0(rw,no_all_squash,root_
squash,sec=krb5:krb5i:krb5p)
```

Nesting Expressions

Just as we can nest commands within the same LINE_MATCHING_CLAUSE, we can also nest expressions by using curly braces {}.

The syntax is as follows:

```
/LINE_MATCHING_CLAUSE/ {/LINE_MATCHING_CLAUSE/ statement}
```

Of course you can nest multiple times as needed:

```
/LINE_MATCHING_CLAUSE/ {/LINE_MATCHING_CLAUSE/ {/LINE_MATCHING_CLAUSE/
statement}}
```

The behavior is like a nested if block. This means that the statement will only run if

- All `LINE_MATCHING_CLAUSE` conditions match
- Because of the nesting, the outer `LINE_MATCHING_CLAUSE` must match before the inner ones are evaluated

Using Multiple Expressions

Sometimes we want to specify multiple expressions having the `LINE_MATCHING_CLAUSE` separately evaluated, avoiding that an unmatched clause prevents the other expressions from running.

As we already saw, besides supplying multiple expressions using the `-e` parameter for each of them, you can also simply list them separated by a semicolon (`;`).

```
/LINE_MATCHING_CLAUSE/ statement;/LINE_MATCHING_CLAUSE/ statement;
```

Line-Matching Clauses

This kind of clause lets you specify per-line matching criteria that can either be a line number or a regular expression.

Note For the reader's convenience, to avoid confusion caused by having `sed` print the whole stream with the changes, in all the following examples, we intentionally disable output to `stdout`(`-n` option) and use the `p` command to print only the lines that match.

Line Matching by Line Number

This is the first and most straightforward line-matching clause: for example, the following expression prints the seventh line of the `/usr/share/doc/curl/README` file:

```
sed -n '7p' /usr/share/doc/curl/README
```

where

- `7` is the line number
- `p` is the print command that causes `sed` to print the contents of the line

The output is as follows:

```
README
```

Line Matching by Regular Expression Pattern

More often we want to focus on lines that match a specific pattern – we already saw this use case in action:

```
sed -n '/ \/tmp / s/defaults/defaults,noexec,nodev,nosuid/p' /etc/fstab
```

`sed` runs the `s` command only on lines matching the " `\/tmp` " RegEx – the string enclosed between slashes (`/`) is a regular expression.

Note Beware that matching is case-sensitive: if you do not need to distinguish between uppercase and lowercase, you must specify the `I` option right after the last slash enclosing the pattern. So, the pattern would become: `/ \/tmp /I`.

Line Interval Matching Clauses

This kind of clause lets you specify the begin and the end criteria that identify the interval of interesting lines: please mind that the begin and the end criteria can either be the line number or a regular expression pattern.

Note The begin and the end criteria that identify the interval of interesting lines are separated by a comma character (`,`).

Line Matching by an Interval of Line Numbers

This is the most straightforward clause we can use to match an interval of lines: simply specify the number of the line where the interval begins and where the interval ends.

For example, to match the lines from one to six

```
sed -n '1,6p' /usr/share/doc/curl/README
```

where

- `1,6` is the interval
- `p` is the print command, that causes sed to print the contents of the line

The output is as follows:

```
      _ _ ____  _
  ___| | | _ \ | |
 / __| | | |_) | |
| (__| |_| | _<| |___
 \___|\___/|_| \_\_____|
```

Line Matching by Start and End Markers

This applies when we want to focus on a marker-based interval: it is a quite common use case when having to configure files structured into stanzas, such as `INI` files.

As a real use case example, let's assume we want to modify `Nobody-User` and `Nobody-Group` values into the `[Mapping]` stanza of the `/etc/idmapd.conf` file so that they are both set to `nfsnobody`.

Note The `/etc/idmapd.conf` file is provided by the `libnfsidmap` RPM package. You can install it by running `sudo dnf install -y libnfsidmap`.

Let's start by figuring out the proper matching clause that prints the lines that we are looking to modify:

```
sed -n '/\[Mapping\]/,/^\[/p' /etc/idmapd.conf
```

The output is as follows:

```
[Mapping]

#Nobody-User = nobody
#Nobody-Group = nobody

[Translation]
```

Since the output looks as expected, the matching clause is correct.

We can now add the substitution commands, completing the `sed` expression. While experimenting, we can put these substitution commands right before the print command so as to have the outcome printed to standard output, allowing us to quickly verify that it works as expected.

The statement is

```
sed -n '/\[Mapping\]/,/^\[/{s/^[ ]*#[ ]*Nobody-User[ ]*=.*/Nobody-User =
nfsnobody/;s/^[ ]*#[ ]*Nobody-Group[ ]*=.*/Nobody-Group = nfsnobody/;p}' /
etc/idmapd.conf
```

The output is as follows:

```
[Mapping]

Nobody-User = nfsnobody
Nobody-Group = nfsnobody

[Translation]
```

This is exactly what we want to achieve, so our trials end here: we found both the right matching clause and the right expression.

Now, if we really want to go on and modify the /etc/idmapd.conf file, we must turn it into

```
sed -i.bak '/\[Mapping\]/,/^\[/{s/^[ ]*#[ ]*Nobody-User[ ]*=.*/Nobody-User
= nfsnobody/;s/^[ ]*#[ ]*Nobody-Group[ ]*=.*/Nobody-Group = nfsnobody/}' /
etc/idmapd.conf
```

Line Matching from a Marker to the End of the File

You may need a line interval spanning from a marker up to the end of the file: simply specify $ as the end marker.

For example, to start matching in the /etc/idmapd.conf file as soon as the [Static] stanza is reached until the end of the file

```
sed -n '/\[Static\]/,$ p' /etc/idmapd.conf
```

Match an Interval of Lines Mixing Line Number and Marker

This applies to only these two use cases:

- Select an interval of lines from a line number to a marker.
- Select an interval of lines from a marker to a line number.

A straightforward use case is picking the interval of lines from the first line to a marker.

From the Beginning of the File to a Marker

To start processing from the beginning of the file until a specific marker is reached, just specify the number 1 as the beginning, and the marker's pattern as the end.

For example, to match from the beginning of the /etc/idmapd.conf file up to the [Static] word that delimits the beginning of the Static stanza

```
sed -n '1,/\[Static\]/ p' /etc/idmapd.conf
```

Negating the Line-Matching Clause

If needed, it is possible to start processing when the matching clause is false by prepending an exclamation mark (!) to the command – we'll see this in action when describing the delete (d) command later.

Sed Commands – The Most Used Ones

After getting familiar with the matching clauses used to trigger commands, we can move forward and explore the most used `sed` commands.

Note Since the following examples are real use cases, I intentionally omit the `-i` parameter to avoid readers running them by mistake without having a clear understanding of the impact of the changes. Of course, if you instead really want to have the statements modify the original files, you must add the `-i` option and use `sudo` if administrative rights are needed.

Append (a)

The lowercase `a` command is used to append the text after the lines that match `LINE_MATCHING_CLAUSE`. For example:

```
sudo sed '/Subsystem[ ]*.*/a \\nMatch User foosvc\n PermitTTY no\n
X11Forwarding no' /etc/ssh/sshd_config
```

It adds a match rule to the SSHd configuration file with directives that prevent the `foosvc` user to set a TTY or to request `X11Forwarding`.

Insert (i)

The lowercase `i` command is used to insert the supplied text before the lines that match `LINE_MATCHING_CLAUSE`. The following example adds the `pam_time.so` module to `/etc/pam.d/login`:

```
sed '/^[ ]*account[ ]*required[ ]*/i account required pam_time.so' /etc/
pam.d/login
```

Note The outcome is enabling time-based logins – hint: if you really want to set this you must also configure `/etc/security/time.conf`.

Change (c)

The lowercase c command is used to replace the whole lines that match LINE_MATCHING_CLAUSE with the supplied text. The following example replaces the parameters supplied to the password quality check pam module (pam_pwquality.so):

```
sed '/password[ ]*.*pam_pwquality\.so/c password requisite pam_pwquality.so
try_first_pass local_users_only retry=3 minlen=8 minclass=3 authtok_type' /
etc/pam.d/system-auth
```

Note This pam module is often configured to set the passwd command to reject new passwords that do not meet the corporate password policy.

Delete (d)

The lowercase d command is used to delete the lines that match LINE_MATCHING_CLAUSE. The following example removes pam_xauth.so from the "/etc/pam.d/su file, thus disabling the automatic forwarding of xauth-keys from the switching user to the user switched to.

```
sed '/pam_xauth\.so/d' /etc/pam.d/su
```

It is of course possible to trigger the delete command only when a line does not match the LINE_MATCHING_CLAUSE: for example, let's pretend that in our lab we have the same fictional NFS exports defined in the "/etc/exports.d/devsecopsbook.exports" file with the following contents:

```
/srv/nfs/foo 192.168.0.0/255.255.255.0(rw,root_squash,sec=krb5:krb5i:krb5p)
/srv/nfs/bar 192.168.0.0/255.255.255.0(rw,root_squash,sec=krb5:krb5i:krb5p)
/srv/nfs/baz 192.168.0.0/255.255.255.0(rw,root_squash,sec=krb5:krb5i:krb5p)
```

and we need to delete all the lines but the /srv/nfs/bar, to preserve it as the only NFS share. The sed expression to achieve this is

```
sed '/^\/srv\/nfs\/bar / !d' /etc/exports.d/devsecopsbook.exports
```

Note See how the exclamation mark (`!`) triggers the command only when the `LINE_MATCHING_CLAUSE` is false.

Substitute (s)

The lowercase `s` command is used to substitute the text matching the substitute pattern in the lines that match `LINE_MATCHING_CLAUSE`.

It requires some arguments using the following syntax:

```
s/regex/replacement/flags
```

Mind that

- The slash character (`/`) is the default delimiter. This can be a little bit unhandy if any of the parameters contain backslashes, since it would require you to escape these backlashes to avoid confusion with the ones used as delimiters. You can overcome this by simply specifying another delimiter such as the plus (`+`) character.
- `regex` is the regular expression used to find the text to be replaced inside the line.
- `replacement` is the text that replaces the matched text.
- `flags` are optional flags that can be specified to alter `s` command default behavior.

Examples of flags are shown in Table 9-3.

Table 9-3. *Sed substitute's command flags*

Flag	Description	Example
i	Enable case-insensitive matching of the supplied RegEx.	`echo ${STRING}|sed 's/oNe/1/i'`
g	Apply to all the matches: the default behavior is to apply only to the first match.	`echo ${STRING}|sed 's/one/1/g'`
number	Skip the first number of matches.	`echo ${STRING}|sed 's/one/1/2'`

The following example locates the line used to implicitly trust users in the `wheel` group: since this setting is commented by default, the statement replaces the # character with nothing, with the outcome getting the line uncommented:

```
sed '/pam_wheel.so trust/s/^[ ]*#//' /etc/pam.d/su
```

Backreferences

We already discussed capture groups and backreferences in the context of regular expressions. As a practical example to experiment with, let's suppose we want to add `minlen=8` to the existing options supplied to `pam_pwquality`.

The `sed` expression to use is

```
sed '/password[ ]*.*pam_pwquality\.so/s/\(.*\)*/\1 minlen=8/' /etc/pam.d/
system-auth
```

Let's raise the stick a little bit and move to a more complex example. Suppose we want to uncomment every NFSv4 directive defined after the `UMICH_SCHEMA` stanza of `/etc/idmapd.conf` file – let's begin by figuring out an expression that fit our needs:

```
sed -n '/\[UMICH_SCHEMA\]/,$ s/#\(NFSv4.*\)/\1/p' /etc/idmapd.conf
```

The outcome is

```
NFSv4_person_objectclass = NFSv4RemotePerson
NFSv4_name_attr = NFSv4Name
NFSv4_uid_attr = UIDNumber
NFSv4_acctname_attr = uid
NFSv4_group_objectclass = NFSv4RemoteGroup
NFSv4_gid_attr = GIDNumber
NFSv4_group_attr = NFSv4Name
NFSv4_member_attr = memberUID
```

Since the outcome looks good, if we want, we can apply the change as follows:

```
sudo sed -i '/\[UMICH_SCHEMA\]/,$ s/#\(NFSv4.*\)/\1/' /etc/idmapd.conf
```

Look at the `/etc/idmapd.conf` file: every NFSv4 directive of that stanza must be uncommented now.

Note Mind that `sed` is limited to no more than nine capture groups.

Summary

In this chapter, we explored stream processing using `sed`. Through various examples, we demonstrated how to perform complex operations by utilizing sed's line-oriented design, including pattern space and hold space. We also learned to restrict processing to specific line intervals to avoid accidental modifications.

Sed is a powerful and efficient tool that remains relevant today, making mastery of its features a significant advantage for DevOps and DevSecOps professionals. It enables efficient text processing and manipulation within scripts and automation workflows, particularly when managing configuration files, logs, and codebases. Its ability to handle large volumes of data quickly makes it invaluable for automating repetitive tasks, reducing the risk of human error and enhancing productivity. Additionally, `sed` can be seamlessly integrated into shell scripts, allowing professionals to create robust automation solutions.

Mastering `sed` streamlines workflows and empowers practitioners to maintain high standards of quality and consistency, ultimately contributing to more reliable and secure software delivery.

CHAPTER 10

Working with AWK

AWK is a powerful pattern scanning and processing language developed by Alfred Aho, Peter Weinberger, and Brian Kernighan at Bell Labs.

The name "AWK" is derived from the initials of their surnames, reflecting the collaborative nature of its creation. Its usage is so widespread that it is nearly impossible to find automation scripts without it.

Although writing AWK scripts may seem anachronistic in today's programming landscape, its concise syntax and handy built-in capabilities remain valuable for crafting effective one-liners. In many cases, it is quicker to code an AWK one-liner than to implement the same logic in a more verbose language like Python.

These one-liners can be executed within shell scripts or run as standalone commands to quickly accomplish operational tasks. AWK excels in typical scenarios such as log parsing, content filtering, and reformatting, making it an excellent tool for integration into CI/CD pipelines.

As a classic tool that remains relevant in modern environments, proficiency in AWK enhances the overall efficiency of DevOps practices, allowing for quicker responses to operational needs and improved data handling in various automation scenarios.

The aim of this chapter is to provide a quick yet comprehensive walkthrough of AWK, focusing on how to write one-liners that can be embedded in shell scripts or used to address common system administration tasks effectively. By mastering these techniques, professionals can enhance their productivity and streamline their workflows.

Note Before going any further, create this **chapter's workspace directory** by running `mkdir -p ~/devsecops-pillars/ch10`, and then immediately navigate to it by running `cd ~/devsecops-pillars/ch10`.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_10

Acquainting to AWK

Conversely from `sed`, which is simply "a utility," AWK is a line-oriented pattern scanning and processing language – this means that AWK can process streams of textual data using statements containing conditional and loop blocks. Just like `sed`, in addition to using plain-text pattern matching, AWK can extensively use regular expressions.

Note The aim of this chapter is to show AWK usage in common use case scenarios that you are likely to find in your daily work: if you want to learn more about this tool, you can refer to its official manual.

Before starting, let's create the `gods-of-it.csv` file with the following contents:

```
Dennis,Ritchie,UNIX and C
Ken,Thompson,UNIX and C
Bjarne,Stroustrup,C++
Richard,Stallman,GNU
Timothy John,Berners-Lee,World Wide Web
Linus,Torvalds,Linux
Theo,de Raadt,OpenSSH and OpenBSD and NetBSD
Phil,Zimmermann,PGP
Brian Jhan,Fox,GNU BASH
Larry,Wall,Perl
Guido,Van Rossum,Python
```

We are going to use this file to do some basic hands-on with AWK.

Note Since we are going to process the file, take care of inserting exactly the above contents, or the outcome of the example statements will be different than described. For example, take care not to have initial or ending blank lines, unnecessary tabs or spaces, and so on.

Fields Variables

When processing a line (often called a "record"), AWK splits it into fields by tokenizing the string using the field separator (the default one is the space ' ' character) and assigning each field to a special variable referenced by the `$` character followed by a number – so

- `$1` is the first field
- `$2` is the second field

and so on.
The whole line itself is stored in the `$0` variable.

Note Mind that, if necessary, you can set a different field separator using the `-F` command-line option.

Shell Invocation

The most common invocation of AWK is using a one-liner. As an example, look at the output of the `lsscsi` command:

```
[0:0:0:0] disk ATA VBOX HARDDISK 1.0 /dev/sda
[0:0:1:0] disk ATA VBOX HARDDISK 1.0 /dev/sdb
```

We can use an AWK one-liner to easily get the disk device with SCSI path `0:0:0:0` as follows:

```
lsscsi | awk '{if($1=="[0:0:0:0]") print $7 }'
```

On my system the output is

```
/dev/sda
```

AWK reads the piped contents line by line: the above AWK statement is just an `if` block satisfied when the contents of the first field of the line are `[0:0:0:0]`. When this condition is met, AWK prints the contents of the seventh field.

One-liners are suitable only for quite short statements typically required to address system operations needs or when embedding AWK into shell scripts. Anyway, mind that AWK was designed to handle much more complex scenarios as well. For more intricate tasks, it is common to load statements from a script file, allowing for better organization, readability, and maintainability of the code.

As an example, let's create the `foo.awk` statement file with the same AWK statement we just ran as a one-liner:

```
{
    if ( $1=="[0:0:0:0]" ) print $7
}
```

Note Since `.awk` files can be recognized in IDEs for syntax highlighting and some additional help, when writing pure AWK scripts, it is better to use the `.awk` suffix to have them automatically turn on the AWK syntax checker.

To have AWK load the statements from the file, launch it specifying the `-f` option followed by the path to the statements file to load. For example:

```
lsscsi | awk -f foo.awk
```

The output is still

```
/dev/sda
```

If necessary, you can even specify the `-f` option multiple times to process more than just one statement file within the same AWK run.

Note Statement files are processed in the same order as they are specified in the command line.

AWK Basic Syntax

Now that we understood AWK's purpose, the time has come to delve into it starting from learning the basic syntax for writing statements.

Pattern-Action Statement

AWK processes a set of

- Pattern-action statements
- Optional function definitions

The most basic structure of an AWK script has the following syntax:

```
pattern { actions }
```

AWK reads the input line by line, checking each time to see if it matches the specified pattern.

The actions are performed when

- The matching is true
- The pattern is omitted (which is logically equivalent to an always matching pattern)

AWK also allows you to

- Specify more than one pattern-action statement
- Define actions that must be performed before beginning to read the lines by using the `BEGIN` pattern keyword
- Define actions that must be performed after reading all the lines by using the `END` pattern keyword

Thus, the full structure of an AWK script can be represented as follows:

```
BEGIN pattern { actions }
pattern { actions }
pattern { actions }
...
pattern { actions }
END pattern { actions }
PATTERNS
```

Patterns

Patterns are expressions that must match the actions triggered.

The default action is printing the matching line, so running `awk` specifying only the pattern makes it behave a little bit like `grep`. For example:

```
awk "/^daemon:|^root:/" /etc/passwd
```

Line-Matching Pattern

When using a line-matching pattern, AWK checks for matches across the entire line. A practical use case for this is checking if there are any filesystems marked to be checked by the `fsck` utility first. This can be detected by examining the entries in the `/etc/fstab` file ending with the number `1`:

```
awk '/ 1\s*$/' /etc/fstab
```

An example match would be as follows:

```
UUID=a62c5b49-755e-41b0-9d36-de3d95e17232 / ext4 defaults 0 1
```

In the above example, the / filesystem is formatted as `ext4` and is marked to be checked by `fsck` first.

Field-Matching Pattern

Field-matching patterns allow for more fine-grained pattern matching targeted to specific input fields. A real-world example of this is checking if there is any filesystem set to be dumped by the `dump` utility. This can be detected by looking for entries in the `/etc/fstab` where the fifth field is set to `1`. While it is possible to use a regular expression for this purpose, a field-matching pattern is often safer, better performing, and cleaner.

An example of such a pattern would be

```
awk '$5=="1"' /etc/fstab
```

An example of a match would be as follows:

```
LABEL=pgsql_data /var/lib/pgsql xfs defaults,noatime 1 0
```

The above example uses an exact match, but it is also possible to handle more complex cases using regular expressions applied to specific fields.

A relevant example is checking if there are any filesystems configured to be mounted with the `noatime` mount option – mount options are specified using a string of comma-separated words in the fourth field of the `/etc/fstab` file.

A regular expression like `,*noatime,*` serves as a starting point, but it is still too generic, since the `noatime` word can appear at the beginning or at the end of the mount options string. To ensure accurate matching, we must refine it including `^` and `$` characters.

Thus, the complete AWK statement to use is

```
awk '$4 ~ /(^|,)noatime(,|$)/' /etc/fstab
```

This statement effectively checks for the presence of the `noatime` option in the fourth field, regardless of its position within the string.

Row Number

In this case the matching pattern is the row number. Since the row number is reported by the `NR` variable, to avoid redundancy, we postpone the example to when talking about variables.

Special Patterns

As mentioned, there are two special patterns that "match" before and after processing the lines, rather than during line processing.

Begin

This keyword marks the actions to be executed before starting to read the input lines:

```
BEGIN { actions }
```

You can use this pattern to

- Print a header for the processed output
- Alter the behavior of the line-matching patterns

For example, the following statement sets the matching patterns to be case-insensitive:

```
awk 'BEGIN{IGNORECASE=1}$1 == "RdCenter"' /etc/services
```

End

This keyword marks the actions to be executed after all input lines have been read:

```
END { action }
```

For instance, the following statement uses the END pattern to print the number of lines in the /etc/passwd file:

```
awk 'END{print NR}' /etc/passwd
```

The NR variable contains the number of processed lines – since we are printing it in the end, it reports the number of lines in the file.

The following example instead shows the usage of both BEGIN and END together:

```
awk 'BEGIN {print "Authors of UNIX Operating Systems:"} /NIX/ {print} END
{print "May Dennis rest in peace, we all owe him a lot"}' gods-of-it.csv
```

produces the following output:

```
Authors of UNIX Operating Systems:
Dennis,Ritchie,UNIX and C
Ken,Thompson,UNIX and C
May Dennis rest in peace, we all owe him a lot
```

Negating the Pattern

```
If we need to run actions on lines that do not match the pattern, we can
negate it using the ! character as follows:
!/pattern1/ { actions }
```

For example, to list every entry in /etc/fstab except those for swap partitions, you can use the following command:

```
 awk '!/ swap/' /etc/fstab
```

Please note the above example was about using exact matches – when dealing with regular expressions, you must negate the match itself, so the ! must be put right before the ~ character.

For example, to print every mount point configured in /etc/fstab except for those with the noatime option

```
awk '$4 !~ /(^|,)noatime(,|$)/' /etc/fstab
```

Logical Operators

Sometimes a single pattern is not enough to uniquely identify the lines we want to run actions on, so we need to specify multiple patterns combined with a logical AND or a logical OR.

Logical AND

We may need to run actions only when multiple matching criteria match at the same time. In such a need we must specify multiple patterns bound with a logical AND (&&).

The syntax to use is illustrated by the following snippet:

```
/pattern1/ && /pattern2/ && /pattern3/ {actions}
```

For example, to check in the /etc/fstab file if there are any XFS-formatted filesystems set to be mounted with the noatime option

```
awk '$3 == "xfs" && $4 ~ /(^|,)noatime(,|$)/' /etc/fstab
```

Logical OR

We may need to run actions when any of the matching patterns matches – in this case we must use the || logical OR:

```
/pattern1/ || /pattern2/ || /pattern3/ { actions }
```

For example, to check if there is any XFS-formatted filesystem or swap partitions configured in the /etc/fstab file

```
awk '$3 == "xfs" || $3 == "swap"' /etc/fstab
```

An alternative way to achieve the same outcome is applying a logical OR within a single matching pattern with multiple criteria:

```
/criteria1|criteria2|criteria3/ { actions }
```

Criteria are separated using a single | since, as we saw, this is the logical OR special character in regular expressions. Thus, the previous statement can be rewritten as follows:

```
awk '$3 ~ /xfs|swap/' /etc/fstab
```

The match() Built-In Function

AWK has built-in functions too: for example, the `match(string, regex)` function returns the position in the string where the regular expression (regex) matches.

The function sets the following variables:

- `RSTART`: Number of characters from the beginning of the field where the beginning of the matching pattern is found
- `RLENGHT`: The length of the string that matches

For example:

```
awk -F , 'match($3, /N.*X/) {print $3 " matches at position "RSTART" and is
"RLENGTH" in length"}' gods-of-it.csv
```

returns the following output:

```
UNIX and C matches at position 2 and is 3 in length
UNIX and C matches at position 2 and is 3 in length
```

Note Please note that besides in the "pattern matching" position, the `match()` built-in function can of course be exploited in the "actions" position.

Actions

Actions are statements executed when the search pattern matches. As mentioned, AWK is a basic programming language, and it includes control structures.

Control Statements

AWK has a few basic control structures such as conditionals and loops.

Conditional Blocks

The basic syntax to declare conditional blocks is as follows:

```
if ( ... ) { ... } else if ( ... && ... ) { ... } else if ( ... || ... )
else { ... }
```

Inside a condition, you can specify booleans (`&&` for logical `AND`, and `||` for logical `OR`) and group them using parenthesis.

For example, the following `if` block is true only for entries of the `/etc/fstab` that are both XFS formatted and mounted with the `noatime` option:

```
awk '{ if($3=="xfs" && $4 ~ /(^|,)noatime(,|$)/) { print } }' /etc/fstab
```

AWK also supports the definition of a switch conditional block using the `switch` statement.

The syntax is as follows:

```
switch (expression) {
case value or regular expression:
   case-body
   break
case other_value or regular expression:
   case-body
   break
default:
   default-body
   break
}
```

As a didactic example, this is a quite unusual way to print only the lines that are not commented out in the `/etc/fstab` file:

```
awk '{ switch($0) { case /#.*/: break; default: print; break; } }' /
etc/fstab
```

Loops

It is also possible to define loops.

Finite Loop

You can create a `for` loop using the following syntax:

```
for (initialization; condition; increment/decrement) {
   statements
}
```

For example:

```
echo iteration | awk '{ for (i=0; i<7; i++) { print $0" "i } }'
```

If you need a "for-each-element-in-list" statement, the syntax is

```
for (index in array) {
   print element[index]
}
```

As in many other languages, you can specify the

- `break` keyword to immediately exit the loop
- `continue` keyword to skip to the next iteration of the loop

Conditional Loop

This is the syntax to be used for a conditional loop:

```
do { statements } while ( condition )
```

For example, the following loop is executed until `i` is lower than three, then it exits the loop and prints the value of `i`:

```
echo iteration | awk '{ i=0; do { i++ } while (i<3); print $0" "i }'
```

If you need an infinite loop, the syntax is

```
do { statements } while (1)
```

If you need to check the condition before executing the loop, you can use a `while` loop using the following syntax:

```
while ( condition ) { statements }
```

Exit

The `exit` statement causes AWK to terminate exiting to the shell who launched it.

Return

The `return` statement is used inside a custom function to terminate the function and return a value. See the "Custom Functions" section for more information on this topic.

Variables

Like every programming language, AWK does allow you to declare variables.

Declare Variables from the Command Line

To declare a variable from the command line, specify it along with its value using the `-v` command-line option.

If you need to specify multiple variables, just repeat this for every variable.

For example, to filter from the `gods-of-it.csv` file only the `Gods of UNIX and C`

```
awk -F ',' -v FILTER='UNIX and C' '{ if($3==FILTER) {print $1" "$2} }'
gods-of-it.csv
```

or, again, to discover who was the father of the World Wide Web

```
awk -F ',' -v FILTER='World Wide Web' '{ if($3==FILTER) {print $1" "$2} }'
gods-of-it.csv
```

Declare Variables Within Statements

You can also declare variables within the AWK statements themselves – the syntax is as follows:

```
variable_name=value
```

As in every programming language, we can also assign to a variable the value of another variable.

For example, we can easily swap two fields of the `gods-of-it.csv` by declaring the `swap` variable and reassigning the value of the fields as follows:

```
awk -F , '{swap=$1; $1=$2; $2=swap; print $1", "$2}' gods-of-it.csv
```

The above one-liner has no pattern to match (so it matches every line of input) and specifies the following actions:

- `swap=$1` assigns the value of the first field to the `swap` variable.
- `$1=$2` assigns the value of the second field to the variable of the first field.
- `$2=swap` assigns the value of the `swap` variable (that was the old value of the first field) to the second field.
- `print $1", "$2` prints the two fields, with their new swapped values.

Note Mind how AWK, conversely from the shells, does not expand variables inside double quotes. For example, `"$3 value is $4"` does not work in AWK.

Incrementing and Decrementing Values

When dealing with numeric values, you can increment or decrement them as follows:

- `var++`: Increment the value of var by 1
- `var--`: Decrement the value of var by 1
- `var+=5`: Increment the value of var by 5
- `var-=5`: Decrement the value of var by 5
- `var*=2`: Multiply the value by 2
- `var/=2`: Divide the value by 2

Modulus

The modulus operator is the `%` character. For example:

```
remainder = 10 % 3
```

sets the remainder of the division of ten by three into the `remainder` variable.

Arrays

AWK does support arrays.

The following snippet illustrates how to declare and add members to an array:

```
array[index]=value
```

For example, this one-liner populates the `info` array and prints its contents:

```
awk 'BEGIN {info[0] = "Marco Antonio"; info[1] = "Carcano"; info[2] =
"DevOps"; for (i in info) {print info[i]}}'
```

The output is

```
Marco Antonio
Carcano
DevOps
```

If you need to remove an element from an array, use the `delete` keyword.

For example, to delete the third item from the `info` array:

```
delete info[3]
```

As an example, let's see how to split the IP address of the `eth0` interface into four octets and finally delete the last octet:

```
ip -4 a show eth0 |grep inet | awk '{ sub("/.*","",$2);
len=split($2,octets,"."); delete octets[4]; for (i=1;i<len;i++) {print
"octet["i"]:"octets[i]} }'
```

The output is

```
octet[1]:10
octet[2]:1
octet[3]:0
```

Note Of course replace `eth0` with the name of a valid interface on your system – you can easily list them by running `ip -a`.

Built-In Variables

AWK has also a few built-in variables that can be exploited when necessary.

ARGV[N]

It is an array containing the files specified in the command line: the first element, `0`, is the `awk` executable itself, the second is the first file specified to be processed, and so on.

For example, the command

```
awk -F ',' -v FILTER='World Wide Web' 'END { print ARGV[0]" "ARGV[1] }'
gods-of-it.csv
```

sets the following variables:

```
ARGV[0] = awk
ARGV[1] = gods-of-it.csv
```

ARGC

This variable contains the length of the `ARGV` array.

ENVIRON

This is an associative array containing the values of the environment variables that were set in the shell before launching `awk`. For example, to print the value of the `PWD` environment variable

```
awk 'BEGIN {print ENVIRON["PWD"]}'
```

NR

It contains the number of the line that is currently processed. For example, printing the number of the line along with its contents

```
awk '{print NR" "$0}' gods-of-it.csv
```

We can exploit the `NR` variable also as a matching pattern. For example, to print the third line from the `gods-of-it.csv` file

```
awk -F ',' 'NR == 3 {print $0}' gods-of-it.csv
```

FS

It contains the `Field Separator`: you can of course alter it as needed. For example:

```
awk -v FILTER='UNIX and C' 'BEGIN {FS=","}{ if($3==FILTER) print $1" "$2 }'
gods-of-it.csv
```

Note Mind that the field separator variable can be set also simply using the `-F` command-line option.

RS

It contains the `Record Separator`, which defaults to the newline character. We can of course change it as needed. In the following example we set it to the # character:

```
echo "Dennis Ritchie,UNIX and C#Ken Thompson,UNIX and C" | awk -v RS=#
'{print}'
```

The output is two different lines, as follows:

```
Dennis Ritchie,UNIX and C
Ken Thompson,UNIX and C
```

NF

It contains the number of fields in the current input record: you can use it, for example, to guess the number of fields in the file.

```
awk -F ',' 'NR == 1 {print NF}' gods-of-it.csv
```

Note how we specified `NR == 1` as a matching pattern, to limit the action of printing the number of fields to be executed only once, when processing the first line.

AWK Functions

AWK provides a lot of functions, but there's not enough room to explain them thoroughly as they would deserve within a few paragraphs: I'm just showing a few examples of the most used functions and how to invoke them.

Note Mind that functions can be used either in "pattern" or in "action" position: this means that the outcome of a function can be used as a matching criterion for the pattern part of that statement.

Generate a Random Number

A common need that sometimes pops up is generating random numbers: for this purpose, AWK provides the `rand` function

```
awk -v seed=${RANDOM} 'BEGIN {srand(seed); print rand()}'
```

In my invocation, the output is

```
0.689777
```

Convert to Lowercase or Uppercase

The `tolower(str)` function can be used to convert a string to lowercase. For example:

```
echo "Hello" | awk '{print tolower($1)}'
```

The output is

```
hello
```

Conversely, we can exploit the `toupper(str)` function to convert a string to uppercase. For example:

```
echo "Hello" | awk '{print toupper($1)}'
```

The output is

```
HELLO
```

Case-Insensitive Comparison

When necessary, we can exploit either the `toupper` or the `tolower` functions also to do case insensitive string comparisons.

For example, to get the line of the RdCenter service from the `/etc/services` file

```
awk '{if(toupper($1)==toupper("RdCenter")) {print}}' /etc/services
```

On my system, the output is

```
RDCenter   4756/tcp       # Reticle Decision Center
```

Substring

A very common requirement is to extract a substring from a string: this can be achieved by using the `substr(str, first, length)` function.

The following example prints a substring of the third field from the beginning of the field (character 0) of four characters in length:

```
awk -F , '$3 ~ /U.*X/ {print substr($3, 0, 4)}' gods-of-it.csv
```

The output is

```
UNIX
UNIX
```

We can of course exploit this function also in conditional statements, for example:

```
awk -F , '{if(substr($3, 0, 4)=="UNIX") {print}}' gods-of-it.csv
```

prints the whole row if the substring of the third field from the beginning of the field (character 0) of four characters in length exactly matches `UNIX`.

The output is

```
Dennis,Ritchie,UNIX and C
Ken,Thompson,UNIX and C
```

Length of Strings

Another common need may be getting the length of a string: this is achieved using the `length(str)` function.

For example, to print the length of the names contained in each of the rows of the `gods-of-it.csv` file (the first field)

```
awk -F , '{print "the length of \""$1"\" is "length($1)}' gods-of-it.csv
```

Since we have not specified any pattern to be matched, the above one-liner prints the length of the names for every record.

The following example shows how to use the `length(str)` as a matching criterion for the pattern part of that statement to print the length of the names (the first field) only when they are longer than six characters:

```
awk -F , 'length($1) > 6 {print "the length of \""$1"\" is "length($1)}'
gods-of-it.csv
```

The output is

```
the length of "Richard" is 7
the length of "Timothy John" is 12
the length of "Brian Jhan" is 10
```

Splitting (Tokenizing) into an Array

As we previously saw, AWK does support arrays. If necessary, we can split a string into tokens that are members of an array using the `split(str, array, regex)` function as follows:

```
awk -F : '$1 == "nobody" {len=split($0,arr,":");for (i=1;i<=len;i++) {print
arr[i]}}' /etc/passwd
```

As you can see, the `split` function returns the number of items resulting from the split – then we use it in the printing `for` loop to put a cap to the iteration.

Sorting an Array

You can sort an array using the `asort(source_array [, destination_array [, sorting_criteria] ])` function: it sorts the array relying on the value of the items.

Note If you need to sort the array based on the indices, you can use the `asorti` function.

Please be wary that

- If `destination_array` is not specified, `source_array` gets overwritten by the outcome of the sort operation
- Sorting criteria, if specified, can be a custom function that implements customized sorting
- Sorting honors the value of the `IGNORECASE` special variable – we already covered it in the "Special Patterns" section.

For example, to load the names of the `gods-of-it.csv` into an array, sort it and finally print it; just issue

```
awk -F , '{names[NR]=$1} END{asort(names); for (name in names) l++; for
(i=1;i<=l;i++) {print names[i]}}' gods-of-it.csv
```

The output is

```
Bjarne
Brian Jhan
Dennis
Guido
Ken
Larry
Linus
Phil
Richard
Theo
Timothy John
```

For more information on this function, please refer to the AWK official manual.

Strings Substitution

Another very common use case is replacing a substring. To achieve that, AWK provides the following functions:

- sub(regex, substitution, string)
- gsub(regex, substitution, string)

The gsub function is a variant of the sub function that performs a global substitution. For example, let's say you have an entry like this in the /etc/passwd file:

```
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

and you want to turn the description FTP User into FTP Chrooted User; simply issue

```
awk -F : '$1 == "ftp" { sub(" User"," Chrooted User",$5); print $1":"$2":
"$3":"$4":"$5":"$6":"$7}' /etc/passwd
```

The output indeed is

```
ftp:x:14:50:FTP Chrooted User:/var/ftp:/sbin/nologin
```

Time Functions

Sometimes it is needed to be able to format or to manipulate time values: for example, you may need to write a condition to operate on records or fields that match a specific date, time, or timestamp.

AWK provides the following time functions:

- mktime(datespec [, utc-flag]): Turn datespec into a timestamp in the same form as is returned by systime(). It is like the function of the same name in ISO C. The argument, datespec, is a string of the form YYYY MM DD HH MM SS [DST].
- strftime([format [, timestamp [, utc-flag]]]): Format the time specified by timestamp based on the contents of the format string and return the result. It is like the function of the same name in ISO C.
- systime(): Return the current time as the number of seconds since the system epoch.

For example, to print the current date and time using AWK

```
LC_ALL=en_US.UTF-8 awk 'BEGIN {print strftime("Today is %B %d, %Y
%H:%M:%S", systime())}'
```

The output is

```
Today is June 14, 2025 12:54:32
```

Running a Shell Command

Running a shell command within AWK is certainly a corner case, but it is not completely unlikely to happen. In such a scenario, you must use the `system(command)` function.

In the following example, we check the status of the `sshd` systemd unit, storing the return code in the `ret` variable:

```
awk 'BEGIN { ret = system("systemctl status sshd"); print "Outcome=" ret }'
```

On my system, the output is

```
● sshd.service - OpenSSH server daemon
     Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; preset:
             enabled)
     Active: active (running) since Sat 2025-01-04 20:29:05 CET; 2 days ago
       Docs: man:sshd(8)
             man:sshd_config(5)
   Main PID: 995 (sshd)
      Tasks: 1 (limit: 10629)
     Memory: 5.1M
        CPU: 161ms
     CGroup: /system.slice/sshd.service
             └─995 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100
               startups"
Outcome=0
```

Note Please mind how the `stdout` of the command is not piped into the `ret` variable that instead gets only the exit code returned by the launched command when it finishes.

Custom Functions

AWK lets you of course define your own functions and call them.

For example, here we define a function that returns if a number is odd or even:

```
awk 'function odd_or_even(num) {
  if (num % 2 == 0) return "even"
  return "odd"
}
BEGIN {
  num=10
  res = odd_or_even(num)
  print num " is " res
}'
```

The output is

```
10 is even
```

Real-Life Examples

Now we learned enough to start unleashing the power of AWK with a few real-life examples: you will probably be surprised to see how many problems it lets you quickly solve with just using one-liners.

Get the Amount of Space Consumed by Old Files

A typical operations task is to free disk space when the system is running out of it. To save money and precious hardware resources, before adding space, the best practice is to check if there are unused files that have been "forgotten" by users and then delete them. A typical task is to identify files that have not been accessed in the last 30 days (important files such as backup files must have already been backed up days before).

The following command calculates the amount of space in MiB used by files that have not been modified for at least 30 days:

```
find /opt -type f -mtime +30 -exec du -sk {} \;|cut -f1 | awk
'{total=total+$1}END{print "Total size is "total/1024" MiB"}'
```

The output is as follows:

```
Total size is 18.332 MiB
```

Processes' Memory Usage

Let's see how to address another problem by using AWK. When developing or even when sizing systems, it is handy to have a detailed report of the memory consumption of specific processes: this is achieved by running the `pmap` command-line utility. For example, to see the memory consumed by the `auditd` process

```
sudo pmap $(pgrep ^auditd)
```

the (cropped) output is

```
828:   /sbin/auditd
0000aaaad7af0000    116K r-x-- auditd
0000aaaad7b1e000      8K r---- auditd
0000aaaad7b20000      4K rw--- auditd
0000aaaad7b21000     16K rw---   [ anon ]
0000aaab0a4c7000    132K rw---   [ anon ]
0000ffffa86b3000     64K -----   [ anon ]
0000ffffa86c3000   8192K rw---   [ anon ]
0000ffffa8ec3000    536K r-x-- libpcre2-8.so.0.11.0
0000ffffa8f49000    100K ----- libpcre2-8.so.0.11.0
...
0000ffffa9819000    140K r-x-- libaudit.so.1.0.0
0000ffffa983c000    112K ----- libaudit.so.1.0.0
0000ffffa9858000      4K r---- libaudit.so.1.0.0
0000ffffa9859000      4K rw--- libaudit.so.1.0.0
0000ffffa985a000     48K rw---   [ anon ]
0000ffffa9866000    164K r-x-- ld-linux-aarch64.so.1
```

```
0000ffffa9895000     32K rw---   [ anon ]
0000ffffa98a1000      8K r----   [ anon ]
0000ffffa98a3000      4K r-x--   [ anon ]
0000ffffa98a4000      8K r---- ld-linux-aarch64.so.1
0000ffffa98a6000      8K rw--- ld-linux-aarch64.so.1
0000fffffbe57000    132K rw---   [ stack ]
total             18756K
```

We can exploit AWK to focus on a specific kind of memory. For example, for the amount of memory used by the process alone

```
sudo pmap $(pgrep ^auditd) | awk '/ auditd/ {total=total+substr($2, 1,
length($2)-1); print} END{print "Total memory used by auditd alone is
"total"Kb"}'
```

The output is

```
0000aaaad7af0000    116K r-x-- auditd
0000aaaad7b1e000      8K r---- auditd
0000aaaad7b20000      4K rw--- auditd
Total memory used by auditd alone is 128Kb
```

In the same way, we can compute the amount of memory used by the process's stack:

```
sudo pmap $(pgrep ^auditd) | awk '/\[ stack \]/ {total=total+substr($2, 1,
length($2)-1); print} END{print "Total memory used by stack is "total"Kb"}'
```

The output is

```
0000fffffbe57000    132K rw---   [ stack ]
Total memory used by stack is 132Kb
```

Or we can compute the amount of anonymous memory used by the process:

```
sudo pmap $(pgrep ^auditd) | awk '$0 ~ /\[ anon \]$/ {total=total+substr($2,
1, length($2)-1); print} END{print "Total anonymous memory is "total"Kb"}'
```

The output is

```
0000aaaad7b21000     16K rw---   [ anon ]
0000aaab0a4c7000    132K rw---   [ anon ]
0000ffffa86b3000     64K -----   [ anon ]
0000ffffa86c3000   8192K rw---   [ anon ]
0000ffffa8f94000      4K rw---   [ anon ]
0000ffffa8fd6000      8K rw---   [ anon ]
0000ffffa8ff9000      8K rw---   [ anon ]
0000ffffa942e000     12K rw---   [ anon ]
0000ffffa9451000      4K rw---   [ anon ]
0000ffffa94c4000      4K rw---   [ anon ]
0000ffffa94e5000      4K rw---   [ anon ]
0000ffffa9688000     48K rw---   [ anon ]
0000ffffa9775000      4K rw---   [ anon ]
0000ffffa9818000      4K rw---   [ anon ]
0000ffffa985a000     48K rw---   [ anon ]
0000ffffa9895000     32K rw---   [ anon ]
0000ffffa98a1000      8K r----   [ anon ]
0000ffffa98a3000      4K r-x--   [ anon ]
Total anonymous memory is 8596Kb
```

And of course we can also compute the amount of memory used by the shared libraries loaded by the process:

```
sudo pmap $(pgrep ^auditd) | awk '$0 !~ /\[ anon \]/ && !/\[ stack \]$/ &&
!/auditd/ {if ($1 !~ /total/) {total=total+substr($2, 1, length($2)-1);
print}} END{print "Total memory used by shared libraries is "total"Kb"}'
```

The last part of the output is

```
0000ffffa97c4000     64K ----- libgssapi_krb5.so.2.2
0000ffffa97d4000      8K r---- libgssapi_krb5.so.2.2
0000ffffa97d6000      4K rw--- libgssapi_krb5.so.2.2
0000ffffa97d7000    132K r-x-- libauparse.so.0.0.0
0000ffffa97f8000    120K ----- libauparse.so.0.0.0
0000ffffa9816000      4K r---- libauparse.so.0.0.0
0000ffffa9817000      4K rw--- libauparse.so.0.0.0
```

```
0000ffffa9819000    140K r-x-- libaudit.so.1.0.0
0000ffffa983c000    112K ----- libaudit.so.1.0.0
0000ffffa9858000      4K r---- libaudit.so.1.0.0
0000ffffa9859000      4K rw--- libaudit.so.1.0.0
0000ffffa9866000    164K r-x-- ld-linux-aarch64.so.1
0000ffffa98a4000      8K r---- ld-linux-aarch64.so.1
0000ffffa98a6000      8K rw--- ld-linux-aarch64.so.1
Total memory used by shared libraries is 9900Kb
```

Summary

In this chapter we explored AWK: as we saw, this pattern scanning and processing language enables users to accomplish more complex tasks compared to `sed`, allowing more complex data manipulation and analysis.

Like `sed`, AWK excels at pattern matching and transformations, making it essential for tasks such as log analysis, configuration management, and data validation. However, AWK goes further with its field-based processing, built-in variables, and support for arithmetic operations, allowing for sophisticated scripting that `sed`'s stream editing alone cannot achieve.

The concepts covered – ranging from basic syntax and one-liners to embedding AWK in shell scripts – equip you to leverage this tool effectively in daily workflows.

You are now aware that while `sed` is ideal for simple text substitutions and line-by-line edits, AWK shines in handling structured data, performing calculations, and generating formatted reports. Together, they form a complementary toolkit: `sed` for quick edits and AWK for deeper processing, automating tasks that reduce manual errors and boost productivity.

For DevOps and DevSecOps professionals, AWK is indispensable in scripts and automation pipelines, since it simplifies data extraction, statistical analysis, and report generation from logs or metrics, enhancing system reliability and insights. Despite their age, AWK and `sed` remain widely used because they offer efficient, lightweight solutions for text and data processing without the overhead of modern languages. Mastering both empowers practitioners to streamline operations and drive data-driven decisions.

PART V

Structured Data Processing

Having familiarized ourselves with basic data formats and the command-line utilities for managing them, we are now ready to explore structured data formats. The most widely used options in this category are `JSON`, `XML`, and `YAML`. Each serves as a powerful tool for data interchange and configuration, providing a clear, organized way to represent structured data.

Proficiency in these formats, combined with practical experience in BASH and Python, equips professionals to write and manage automation scripts and integrations that streamline continuous integration and continuous deployment (CI/CD) workflows. Understanding how to handle these formats enables efficient data management, facilitates seamless integration between tools and systems, and supports robust automation processes. Ultimately, this knowledge fosters more secure and reliable software development practices within the DevOps framework, ensuring applications are built, tested, and deployed efficiently.

In the chapters of this section, we will delve into the details of each format individually, while also exploring their associated command-line tools: `jq` for JSON, `xpath` and `xmlstarlet` for XML, and `yq` for YAML. Mastering these tools is crucial, as they can be integrated into shell scripts and CI/CD pipelines for efficient command-line management of complex data types.

Additionally, we will demonstrate how to process these formats using Python, showing the step-by-step evolution of an application that incorporates support for each format as a distinct feature. This example will also illustrate best practices for structuring an executable Python package, deepening our understanding of both data handling and software organization.

CHAPTER 11

XML with XPath, XMLStarlet, and Python

The Extensible Markup Language (XML) is an essential skill for DevOps (and DevSecOps) professionals, extending far beyond its role as a simple data-interchange format. It underpins many enterprise applications, including configuration files (e.g., Apache Tomcat's `server.xml`, `context.xml`), OSGi Blueprints, JAAS configurations, Maven POM files, and SCAP standards like OVAL and XCCDF. XML also powers protocols such as SOAP (Simple Object Access Protocol) and SAML, as well as description languages like WSDL (Web Services Description Language).

Mastering XML involves understanding Document Type Definitions (DTD) and XML Schema Definitions (XSD), which are critical for validating data structures. This ensures configurations adhere to specified formats, bolstering security and compliance.

Proficiency in XML Path Language (XPath) syntax, along with tools like `xpath` and `xmlstarlet`, enables efficient querying and manipulation of XML data, supporting automation and integration in CI/CD pipelines. Additionally, knowledge of XSLT for transforming XML into HTML enhances reporting and monitoring capabilities. Familiarity with Python's `lxml` library is equally vital, as it allows programmatic processing of XML documents, streamlining workflows and promoting a more secure, efficient environment.

This chapter covers these concepts comprehensively, beginning with an overview of XML, its objectives, and its enforcement through schemas. It also explains generating human-readable XML versions via XSLT. You'll explore XPath with examples using the `xpath` command-line tool, as well as modifying XML nodes and attributes with `xmlstarlet`.

By the end, you'll be equipped to integrate these tools into scripts for XML support. The chapter also demonstrates processing XML in Python using `lxml`, reinforcing practical application in DevOps scenarios.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_11

Note In this chapter we are using the book's companion resources for the first time, which are provided in the `devsecops-pillars.tar.gz` file. After copying the `devsecops-pillars.tar.gz` file to your VM, extract them by running `tar -xzvf devsecops-pillars.tar.gz -C ~/devsecops-pillars`. If you are using the `Vagrantfile` provided with the book supplementary material and you followed the deployment guidelines at the beginning of the book, assuming shared folders are operational, the `devsecops-pillars.tar.gz` must be already available in the `/vagrant` directory. If shared folders are not working, you can copy the gzipped tarball manually using the `vagrant scp` command. For example, `vagrant scp devsecops-pillars.tar.gz devsecops_lab_01:~`.

What Is XML

The eXtensible Markup Language (XML) is a markup language aimed at defining a set of rules for encoding documents both in a human-readable and machine-readable format. The two most important principles of XML are as follows:

- It relies on a well-formed tags-based syntax (all tags are enclosed between < and > and must be closed).
- The document has a hierarchical structure that can be enforced using a schema.

Note Special XML characters such as < and > must be escaped when used within element values.

Table 11-1 summarizes the most used schema languages.

Table 11-1. *Schema languages*

Schema Language	Description
Document Type Definitions (DTDs)	It uses a concise formal syntax that is mostly focused on which attributes an element possesses and on the nesting of the elements. Elements' attributes support only very simple data types.
XML Schema Language	It addresses the same needs of DTD, improving it by adding features such as namespacing, strong data typing, inheritance, and attribute validation constraints.
Relax NG	It is a sort of compromise between the previous two.
Schematron	It is aimed at making assertions about the presence or absence of patterns in XML trees: you can use it in addition to other schema languages.

XML tags are placeholders enclosed in angle brackets used to define elements in an XML document: once opened, a tag is closed by using the forward slash (/) character.

From their closing perspective, they come in two types: `regular` and `self-closed`, as depicted by Table 11-2.

Table 11-2. *Tag types*

Tag Type	Description	Example

regular	They come in pairs: an opening tag and a closing tag; the latter has the identifier preceded by the / closing character. They are used to enclose contents – either text or other nested tags. These latter are also called children.	`<ts>2024-12-01T08:20:13Z</ts>`
		`<alert>`
		`<message>`
		`Free space in /home is less than 10%`
		`</message>`
		`</alert>`

self-closed	They combine the opening and closing tags into a single one, placing the forward slash right before the closing angle bracket. This type of tag has no content other than attributes.	`<foo bar="baz" />`

It is possible to provide additional information by adding attributes to the tags, no matter if they are regular or self-closed ones.

Attributes are name-value pairs enclosed in either single (') or double (") quotes.

For example:

```
<alert severity="CRITICAL">
  <notify command='echo "${message}" | /usr/bin/mail -s "CRITICAL - /home
  below 10%" sysadmins@carcano.corp' />
  <message subject="CRITICAL: Free space on /home is less than 10%">
    Free space in /home is less than 10%
  </message>
</alert>
```

Note When developing your own XML content, nested tags are useful for representing hierarchical data and providing additional information in a structured format. If a flat format is more suitable for your data representation, you may prefer to use attributes instead.

As stated, XML documents can be validated using a schema document.

It is even possible to use tags belonging to different schemas within the same XML document: this promotes document schema reuse, in compliance with the Don't Repeat Yourself (DRY) principle, as you can share schemas across different XML documents.

In such scenarios, each distinct schema is assigned to a proper namespace, and the XML elements (tags) or attributes are scoped into the correct namespace by prepending their specific namespace.

Note It is possible to assign multiple schema documents within the same namespace.

For example:

```
<alert xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="https://www.carcano.corp/schema"
       xsi:schemaLocation="https://www.carcano.corp/schema alert.xsd">
  <c:message>
```

```
    Free space in /home is less than 10%
  </c:message>
</alert>
```

The above snippet can be embedded in an XML document as a child of an XML node having another schema: when the parser needs to validate the alert node, it automatically switches to the correct schema.

A Simple XML File

To make learning easier, we'll use a hands-on approach with examples, starting with the Fancymonitor application. This is a fictional monitoring app that serves as our case study throughout the book.

Since the initial version of Fancymonitor supports XML configuration files, we'll use its provided files as practical examples.

To have a look at them, let's change to the Fancymonitor project directory tree as follows:

```
cd ~/devsecops-pillars/fancymonitor
```

then, change to the `src` directory:

```
cd src
```

Since this is a local Git repository, to operate on it we must install the `git` command-line tool:

```
sudo dnf -y install git
```

Note From this chapter onward, we will often use the `git` command-line tool to compare the changes made to the Fancymonitor project code as specific features are implemented. Git is the most widely used version control software and is an essential tool and skill for professionals. A comprehensive explanation of Git will also involve working with remote repositories; therefore, we will postpone this discussion until Chapter 35, after installing Gitea. For now, just focus on executing the commands provided and familiarize yourself with the brief and targeted information presented in the descriptions leading up to Chapter 35.

Once installed, we must check out the branch containing only the XML configuration files:

```
git checkout f-xml-only
```

Now, let's have a look at the `etc/config.xml` configuration file:

```
cat etc/config.xml
```

It begins with a tag claiming the XML version in use and the character set encoding required to parse the rest of the document:

```
<?xml version="1.0" encoding="UTF-8"?>
```

The actual document is then structured using a set of custom-defined tags (`probes`, `probe`, `description`, `statement`, `alert`, ...): some of them also have attributes – for example, the `probe` tag has the `id` attribute, while others don't.

The following tree highlights the overall structure of this sample XML document:

```
<probes>
   |--<probe>
   | |--<platforms>
   | |   |--<platform>
   | |   \--...
   | |--<description>
   | |--<statement>
   | |   |--<vars>
   | |   |    |--<var>
   | |   |    \--...
   | |   \--<command>
   |  \--<alert>
   |     \--<notify>
   \-- ...
```

As you can see, the document is structured to enable us to declare as many probes as we want and to define notifications when needed.

In the `etc/config.xml` file we are defining only two probes:

- The first probe, which generates an alert with MAJOR severity, is raised when the free disk space on the `/var/log` mount point is less than 20%.
- The second probe, which generates an alert with CRITICAL severity, is raised when the free disk space on the `/var/log` mount point is less than 10%.

Both probes provide the shell command to run to get the current free space and exit, returning `0` if it is above the threshold or returning `1` if it is below the threshold.

The second probe also provides the shell command to run to send an email notification.

Note XML is a huge topic that is not possible to thoroughly explain in this short amount of space: if you need further information, you can read "Structures" (https://www.w3.org/TR/xmlschema11-1/) and "Datatypes" (https://www.w3.org/TR/xmlschema11-2/) recommendations.

Enforcing the XML Syntax with a Schema

XML goes far beyond simply providing a data structure format: it also provides schema enforcement by validating the data structure against one or more schema documents.

This is invaluable: we are humans, and humans have fat fingers and make mistakes – having a quick and easy way to validate documents is often a lifesaver, since after validation we are sure any XML parser can make sense of the document.

Validation is performed using an agnostic XML linter application able to parse any XML document and warn about errors: when using DTD it can warn about syntax errors, so it validates the grammar, the structure of a document, and the object type of the attributes.

Even better, we can even use a more advanced schema definition such as XSD: this format enables us to also describe the validation constraints for the values of the attributes, enabling even to describe an interface that can also validate the provided values.

This means that

- It is possible to use XML with XSD to display a form (either web or a desktop application) by describing not only the controls it must have, but also validation constraints for the text boxes, pre-filling the combo boxes with only valid values, and so on. An example of this is the `scap-workbench` application, which is part of the OpenSCAP suite.
- When developing a web service, it is possible to use the XSD to validate the values of the attributes of the received input.
- If necessary, it is possible to use the XSD to generate a WSDL.

and so on.

A lot of software relies on schema enforcement: as a practical example, have a quick look at OpenSCAP.

OpenSCAP is the open-source implementation of the Security Content Automation Protocol (SCAP), consisting of a scan engine providing benchmarks for running compliance checks.

For our didactic purpose, it is a very good application for learning XML, since it deeply relies on it for configuring its benchmarks: it indeed uses a lot of different documents with different schemas (OVAL, XCCDF, DataStream, …).

Let's install it and have a close look at it – enter the following command:

```
sudo dnf -y install openscap scap-workbench
```

Once installed, besides the scanner itself, the package provides a lot of XML-based documents along with the schema files.

Let's have a closer look at them.

DTD Schemas

This is the list of DTD schema files provided by the OpenSCAP RPM package:

- `/usr/share/openscap/schemas/xccdf/1.1/XMLSchema.dtd`
- `/usr/share/openscap/schemas/xccdf/1.1/datatypes.dtd`
- `/usr/share/openscap/schemas/xccdf/1.2/XMLSchema.dtd`
- `/usr/share/openscap/schemas/xccdf/1.2/datatypes.dtd`

As you can see all of them have names with a trailing `.dtd`.

XML Schemas

This is the initial part of the list of XML schema files provided by the OpenSCAP RPM package:

- `/usr/share/openscap/schemas/oval/5.9/aix-definitions-schema.xsd`
- `/usr/share/openscap/schemas/oval/5.9/aix-system-characteristics-schema.xsd`
- `/usr/share/openscap/schemas/oval/5.9/apache-definitions-schema.xsd`
- `...`

Note Listing all of them here does not really matter: just note that all of them have names ending by `.xsd`.

Schematrons

This is the list of schematrons schema files provided by the OpenSCAP RPM package:

- `/usr/share/openscap/schemas/oval/5.9/oval-definitions-schematron.xsl`
- `/usr/share/openscap/schemas/oval/5.9/oval-directives-schematron.xsl`
- `/usr/share/openscap/schemas/oval/5.9/oval-results-schematron.xsl`
- `/usr/share/openscap/schemas/oval/5.9/oval-system-characteristics-schematron.xsl`
- `/usr/share/openscap/schemas/oval/5.9/oval-variables-schematron.xsl`

As you can see, all of them have names with a trailing `-schematrons.xsl`.

Note To save space in the book, I am intentionally omitting a lot of files: my aim is just providing you directions on where to find these kinds of schema files so that you can have a closer look at them by yourself if you fancy.

Example DTD file

DTD is the most basic schema enforcement mechanism provided by XML (it is using a subset of the SGML DTD).

It is mostly focused on specifying

- What attributes can an element have
- The nesting of the elements
- Very simple data types for the attributes

The Fancymonitor application's `etc/schemas/probes.dtd` file provides the DTD schema supporting the `etc/config.xml`.

Note If you are no longer in the Fancymonitor's `src` directory, change back to it `cd ~/devsecops-pillars/fancymonitor/src`.

Let's open it to have a closer look:

```
cat etc/schemas/probes.dtd
```

The syntax is easy: we just declare elements (`ELEMENT`) and the list of attributes (`ATTLIST`) an element has, taking care of describing the nesting of the elements themselves.

For example:

```
<!ELEMENT alert (notify?) >
<!ATTLIST alert severity CDATA #REQUIRED >
<!ELEMENT notify (#PCDATA) >
```

The above snippet claims that the element `alert` has a required string (`CDATA`) attribute called severity and may have zero or one element `notify` as children; the `notify` element can contain Parsed Character Data (`PCDATA`): this means it can contain any free text.

Note The full DTD specification is available at https://www.w3.org/TR/xhtml1/#dtds.

As per best practice, the `etc/config.xml` document embeds the path to the DTD file, so that XML parsers can know that there's a specific schema they can use to validate the syntax.

This is set by the following line:

```
<!DOCTYPE probes SYSTEM "schemas/probes.dtd">
```

Note In this example we are using a local URI, but in real life it is better to use a remote URI, such as `<!DOCTYPE probes SYSTEM "https://www.carcano.corp/schemas/probes.dtd">`: this way you have a single point for distributing your DTDs.

This means we can validate the XML document by asking the linter to use the DTD schema – in this example, we use `xmllint`, a utility that has been specifically developed to lint XML files:

```
xmllint --noout --valid etc/config.xml
```

No news means good news – the XML document passed the validation.

It may happen that we have to deal with XML files not embedding the DTD schema location – in this case we must provide it explicitly by using the `--dtdvalid` option followed by the path to the DTD file.

For example:

```
xmllint --noout --dtdvalid etc/schemas/probes.dtd etc/config.xml
```

For the sake of completeness, let's try testing a DTD validation failure.

For instance, let's alter the `etc/config.xml` file by adding a `foo` self-closed tag somewhere.

In the below example, I added it at the end of the document, beneath the last closing of the probe and probes tags:

```
  </probe>
  <foo />
</probes>
```

Since the syntax is still well formed, the document is still a valid XML, but since the foo self-closed tag is not defined in the DTD, validation is expected to fail because of a non-compliance found.

Run the validation command again – this time the linter must return an error containing the following message:

```
Element probes content does not follow the DTD, expecting (probe)+, got
(probe probe foo )
```

Leave the etc/config.xml as it is now, do not remove the foo self-closed tag since we need it again to practice with the XSD schema validation.

Example XSD file

The shortcoming with DTD is its missing support for defining constraints: the XML Schema Document (XSD) fills this gap and provides also more granular control on the format of the document.

To see XSD in action, we can have a look at the etc/schemas/probes.xsd file – this is an improvement of the DTD we have just played with.

Let's open it as follows:

```
cat etc/schemas/probes.xsd
```

As you can see, conversely from the DTD, the XSD file is in XML format.

Similarly, as for the DTD file, by best practice, the etc/config.xml document embeds also the path to the XSD file, so that XML parsers can know that there's a specific schema they can use to validate the syntax.

This is set by the following line:

```
<probes xmlns="https://www.carcano.corp" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="https://www.carcano.corp file://../
etc/schemas/probes.xsd">
```

Now let's validate it with `xmllint` – mind anyway that `xmllint` do not support automatic retrieval of the XSD, so we must explicitly pass the path to the XSD file as follows:

```
xmllint --noout --schema etc/schemas/probes.xsd etc/config.xml
```

It fails with a message containing the following two errors:

```
Element '{https://www.carcano.corp}notify', attribute 'trigger_every':
[facet 'minInclusive'] The value '5' is less than the minimum value allowed
('300').
Element '{https://www.carcano.corp}foo': This element is not expected.
Expected is ( {https://www.carcano.corp}probe ).
```

This is expected, since when playing with DTD we added the `foo` self-closed tag specifically for causing this error.

Remove it and re-run the above `xmllint` statement – it will return an error containing only the following message:

```
Element '{https://www.carcano.corp}notify', attribute 'trigger_every':
[facet 'minInclusive'] The value '5' is less than the minimum value allowed
('300').
```

This time we have only the value-related error: this is because, conversely from DTD, XSD also supports advanced value validation rules – you can even use RegEx as matching criteria.

In this specific case, we are hitting against these constraints:

```
 <xsd:attribute name="trigger_every" use="required">
  <xsd:simpleType>
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="300"/>
      <xsd:maxInclusive value="3600"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

They claim that the value of the `trigger_every` attribute must be between `300` and `3600`.

To fix it just change the value in the `etc/config.xml` file from `5` to `300`, then re-run the linter.

This time it must pass the validation, and you must get the following message:

```
etc/config.xml validates
```

The XPath Query Language

The XML Path Language (XPath) is a query language designed to select nodes in XML documents. It is used in various languages and tools such as `XSLT`, `XQuery`, and it is mainly employed in Software Development and Software Testing fields. Since DevOps and DevSecOps professionals cooperate with teams in both fields, then skills on XPath are a must-have for them too.

XPath selects the contents by using nodes as the coordinates of a tree-like document.

Nodes can be

- `Elements`: These nodes are the actual XML tags. They can contain other nodes, including child elements and text nodes and even nodes of the kind `Attribute`. The topmost element node is called the `Root Node`, and it is denoted by the `/` character.
- `Attributes`: These nodes are the property/attribute that `Element` nodes may have.

XPath supports the following data types:

- `Number`: Can be either an integer or floating point (IEEE 754)
- `Boolean`: Either true or false
- `String`: A string of zero or more characters
- `Sequence`: A collection of any kind of node as well as atomic values (like numbers and strings)

When using development languages, XPath-compatible libraries let you select a specific node by providing its XPath: in this case it becomes the `Context Node` you can then use as a reference when providing other XPaths.

Note The `etc/xslt/config.xsl` XSLT file provides several examples of `Context Node` selection: loop iterations are indeed a very common use case for this. We will discuss XSLT in a dedicated paragraph at the end of this chapter.

It is also possible to specify a set of nodes that relates to the `Context Node` in a specific way using `Axes`, as summarized by Table 11-3.

Table 11-3. *XPath Axes*

Axes	Description
child	Selects children of the `Context Node`
parent	Selects the parent of the `Context Node`
ancestor	Selects all ancestors (parents, grandparents, etc.) of the `Context Node`
descendant	Selects all descendants (children, grandchildren, etc.) of the `Context Node`
following-sibling	Selects all siblings after the `Context Node`
preceding-sibling	Selects all siblings before the `Context Node`
self	Selects the `Context Node` itself

The `etc/config.xml` document provides a good example for demonstrating the use of some different `Axes`.

For instance, if the `Context Node` is the first `probe` element

- `child::description` selects its `description` child element
- `parent::probes` selects the `probes` element
- `following-sibling::probe` selects the second `probe` element

In XPath expressions, paths can be expressed either as `Absolute` or `Relative` paths, as summarized by Table 11-4.

Table 11-4. *XPath paths*

Path	Description	Example
Absolute	Begins at the Root node	`/probes`
Relative	The path is relative to the context node. For example, once a `probe` node is selected, it is possible to refer to its child `description` node avoiding to type the full path	`description`

`Attribute` nodes can be accessed by prepending the `@` character to the name of the attribute itself.

For example, if current node is one of the `probe` nodes, we can use the following relative path to get the value of the `severity` attribute of its `alert` child node:

```
alert/@severity
```

When dealing with a list, it may be necessary to select a specific child element of a node. The most straightforward way is selecting it by its index, for example:

```
/probes/probe[1]/statement
```

Note Mind that in XPath indexes are 1-based, so the first index element is 1.

A very common and less trivial need in selecting nodes from a list is picking the one (or ones) having attributes matching specific criteria.

In this case you must use a predicate, which is an expression with one or more conditions enclosed beneath square brackets `[]`.

For example, to select the probe with `id=2`

```
/probes/probe[@id='2']
```

You can of course specify an XPath expression containing a predicate for selecting the child of a node which matches specific criteria.

For example, to select the `description` element of the probe with `id=2`

```
/probes/probe[@id='2']/description
```

The wildcard character in XPath expressions is the star (*): it matches any `Element` node at the current level of the `Context Node`. For example:

```
/probes/probe[1]/*
```

selects any `Element` node child of the first `probe` node.

When dealing with attributes, it is possible to use * and `@` together to match any element having an attribute.

For example:

```
/probes/probe[@*]
```

If necessary, it is possible to specify more complex predicates with criteria matching children of the element to return.

For example, to select the `probe` elements having the `alert` element set to `CRITICAL`

```
/probes/probe[alert[@severity='CRITICAL']]
```

Another example is selecting the `probe` elements having the `alert` element not containing a `notify` element:

```
/probes/probe[alert[not(notify)]]
```

Working with the XPath Command-Line Utility

After having familiarized with the XPath syntax, we can start practicing with it using the `xpath` command-line tool.

This is a utility provided by the `perl-XML-XPath` RPM package that can be used to get the contents of XML nodes selecting them using XPath expressions.

We can install it as follows:

```
sudo dnf install -y perl-XML-XPath
```

As an example, let's extract every `var` element from the first `probe` element:

```
xpath -q -e "/probes/probe[1]/statement/vars/*/text()" etc/config.xml
```

It returns

```
mountpoint=/var/log
threshold=20
```

Since we specified the `/text()` function, the returned node is just text, without the opening `<var>` and closing `</var>` tags.

Note As an exercise, spend some time trying all the XPath examples we have seen so far, and maybe invent some on your own, just to get the hang of this handy tool.

Working with xmlstarlet Command-Line Utility

The `xpath` command-line utility is quite handy, but it is also quite limited in returning the contents in a raw format and, even worse, it cannot be used to edit the XML document.

A more complete tool is `xmlstarlet` – we can install it as follows:

```
sudo dnf install -y xmlstarlet
```

`xmlstarlet` provides much more features than `xpath`, but at the price of being more complex.

For example, it is very picky about specifying the default namespace: more specifically, if the document sets it, we must declare an alias to it in the `xmlstarlet` statements too.

In our case, the `etc/config.xml` document contains the namespace definition for the whole `probes` document:

```
<probes xmlns="https://www.carcano.corp" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="https://www.carcano.corp file://../
etc/schemas/probes.xsd">
```

In this row, the `probes` tag contains the attribute `xmlns` which defines the `https://www.carcano.corp` namespace for the entire `probes` tree.

In such a scenario, `xmlstarlet` requires defining an alias for each namespace in the XPath expression. These aliases can be specified by using the `-N` option.

In our case

```
-N "c=https://www.carcano.corp"
```

assigns the `c` alias to the `https://www.carcano.corp` namespace.

The alias must then be prefixed to each element in the XPath expression.

So, the XPath expression

```
/probes/probe[1]/@id
```

in `xmlstarlet`, after adding the `c` alias, becomes

```
/c:probes/c:probe[1]/@id
```

Let's put everything together and run it as follows:

```
xmlstarlet sel -N "c=https://www.carcano.corp" -t \
-v "/c:probes/c:probe[1]/@id" etc/config.xml
```

The meaning of the command-line options used is as follows:

- `sel` is the `xmlstarlet` option to the `select` function (the one used to select contents).
- `-t` tells `xmlstarlet` to use a template.
- `-v` provides the XPath expression to be used to select the values using the template.

The output is

```
1
```

which indeed is the value of the `id` attribute of the first `probe` element in the list.

Inside shell scripts, we can use the output of `xmlstarlet` statements like the last one we ran to fetch contents and export it as environment variables.

The following example shows how to do this, retrieving the `vars` elements of the `statement` of the `probe` with `id=2` and exporting them as environment variables:

```
export $(xmlstarlet sel -N "c=https://www.carcano.corp" -t -v "/c:probes/
c:probe[@id='2']/c:statement/c:vars/*" ${HOME}/devsecops-pillars/
fancymonitor/src/etc/config.xml)
```

In this example, the output of the `xmlstartlet` statement returned by the subshell is

```
mountpoint=/var/log
threshold=10
```

Because of the `export` command preceding the output, both `mountpoint` and `threshold` are exported as environment variables.

We can easily verify this as follows:

```
echo ${mountpoint}
echo ${threshold}
```

In the same way, we can then retrieve the `command` run by the `probe` as follows:

```
xmlstarlet sel -N "c=https://www.carcano.corp" -t -v "/c:probes/c:probe
[@id='2']/c:statement/c:command" etc/config.xml
```

This returns

```
if [ $(
        df --output=size,avail ${mountpoint}
        | tail -n 1
        | awk '{pfree=$2*100/$1; printf "%3.0f\n", pfree}'
      ) -lt ${threshold}
]; then
  exit 1;
fi
```

For your convenience I put it on multiple lines to improve the readability.

It is a small shell script snippet checking the free disk space: if it is below the `threshold` value, it exits and returns `1`.

These two mechanisms can be exploited by the monitoring application: after fetching and exporting the `vars` as shown in the first example, the application can just fetch the `command` to run and launch it using `eval`

The actual probe can be easily implemented inside a shell script by running the above `xmlstarlet` sentences in sequence:

```
export $(xmlstarlet sel -N "c=https://www.carcano.corp" -t -v "/c:probes/
c:probe[@id='2']/c:statement/c:vars/*" ${HOME}/devsecops-pillars/
fancymonitor/src/etc/config.xml)
eval $(xmlstarlet sel -N "c=https://www.carcano.corp" -t -v "/c:probes/
c:probe[@id='2']/c:statement/c:command" ${HOME}/devsecops-pillars/
fancymonitor/src/etc/config.xml)
```

The script can then check if the exit value of the `eval` statement is `1`: in such an event, since the probe reported a failure, the script must run the code with the necessary actions to be taken. This involves also checking other elements such as `alert` and `notify`, implementing the actual logic accordingly.

Note As you probably are guessing, you can use this `export` / `eval` trick with `xmlstarlet` to write powerful shell scripts. Anyway, mind that if you forecast an extensive use of XML, it is better to use a scripting language such as Python, since they have native libraries for managing the XML format.

The last use case we explore with `xmlstarlet` is editing the XML document.

As an example, we are editing the description of the probe with `id=2` turning it into `Ensures that free disk space is at least 10% and notifies via email when the event is hit`:

The statement to run is

```
xmlstarlet ed --inplace -N "c=https://www.carcano.corp" \
-u "/c:probes/c:probe[@id='2']/c:description" \
-v "Ensures that free disk space is at least 10% and notifies via email
when the event is hit" \
etc/config.xml
```

The meaning of command-line options used is as follows:

- `ed` is the `xmlstarlet` option to `edit`.
- `-u` sets the node to be updated.
- `-v` the new value for the node.

Let's check if it worked:

```
xpath -q -e "/probes/probe[@id='2']/description" etc/config.xml
```

As expected, the output is

```
<description>Ensure that free disk space is at least 10% and notifies via
email when the event is hit</description>
```

Transforming Using XSLT

XSLT (Extensible Stylesheet Language Transformations) is a language for transforming XML documents converting data from one XML schema to another.

Suitable use cases for XSLT are

- Formatting into human-readable documents, such as converting XML into HTML
- Data exchange – converting data from one XML schema to another or into a data exchange format such as SOAP

Besides its own built-in functions, it also leverages the XPath query language to select the subset of the XML document relevant for each specific rendering. This means that with XSLT you can either transform the whole XML document as well as a subset of it.

The conversion requires providing XSLT Templates using a file, called XSLT stylesheet, written in Extensible Stylesheet Language (XSL) templating language. Besides XSL statements, this file may also contain additional information such as file headers and instruction blocks that will be added to the output file.

To show you XSL in action, the Fancymonitor project provides the `etc/xslt/config.xsl` file: it has been developed for enabling the generation of HTML documents from the contents of the `etc/config.xml` file.

Before trying the actual transformation, open the `etc/xslt/config.xsl` and have a look at its contents.

```
cat etc/xslt/config.xsl
```

As you can see, in the same way as `xmlstarlet`, we declared `c` as the alias of the `https://www.carcano.corp` XML namespace, adding the `c` alias as a prefix for each element of the XPath expressions.

XSL stylesheets typically contain several template elements which are referenced by the main template element (called /).

Each template describes the specific transformation to apply to the queries' outcome or when a certain XML element is found in the XML file: in this specific case, each template describes the transformation to HTML process for some specific element.

The `etc/xslt/config.xsl` file contains four templates:

- / (lines 4–35)

- `c:alert` (lines 36–58)
- `c:statement` (lines 59–75)
- `c:probes` (lines 76–102)

These templates are then instantiated inside other templates. For example, the `c:probes` template is instantiated beneath the / template at line 29, the `c:statement` template is instantiated beneath the `c:probes` template at line 98, and the `c:alert` template is instantiated beneath the `c:probes` template at line 99.

Note Please note that template definitions can be listed in any order.

Note how inside templates we make use of XSLT statements such as the `for-each`, using them to iterate over XML nodes. Look at the example in the following snippet that is cut from `etc/xslt/config.xsl` (line 83 and line 100):

```
<xsl:for-each select="c:probe">
<tr>
...
</tr>
</xsl:for-each>
```

Note also how we render the values fetched by the XPath select using `<xsl:value-of select="..."/>` statements.

For example:

```
<xsl:value-of select="@id"/>
```

fetches the value of the `id` attribute of the current node.

Similarly, as for the DTD and XSD files, by best practice, the `etc/config.xml` document embeds also the path to the XSLT file. This is set by the following line:

```
<?xml-stylesheet type="text/xsl" href="xslt/config.xsl"?>
```

Now that we have a basic grasp of XSLT syntax, let's put it into action by generating an HTML document.

In this example, we are generating the `/tmp/config.html` by using `xsltproc` command-line utility – just type the following command:

```
xsltproc -o /tmp/config.html etc/config.xml
```

Open with a web browser the generated HTML file, just to see how this magic really did the trick.

Please note that most browsers can process XSLT on the fly, allowing you to open `etc/config.xml` directly in your favorite browser and have it rendered as HTML. But first, read the following note.

Note Most web browsers consider processing XSLT stored on the local filesystem a security risk: for example, Firefox raises the `CORS request not http` error. You can disable this security mechanism (at your own risk) by navigating to `about:config`, searching for `security.fileuri.stric_origin_policy`, and setting it to `false`.

There's a simple workaround to avoid disabling the security policy: just serve both the XML and XSL files from a temporary HTTP server, then connect to it with your browser and open the XML document through that server.

For this purpose, we can, for example, exploit the HTTP server provided by Python – first we need to change to the directory where the xml file is stored:

```
cd ~/devsecops-pillars/fancymonitor/src/etc
```

If `firewalld` is running, we must also add a temporary exception to connect to port `8080`.

For example, to add a 10-minute exception

```
sudo firewall-cmd --add-port=8080/tcp --timeout=10m
```

We can now launch the Python `http` module:

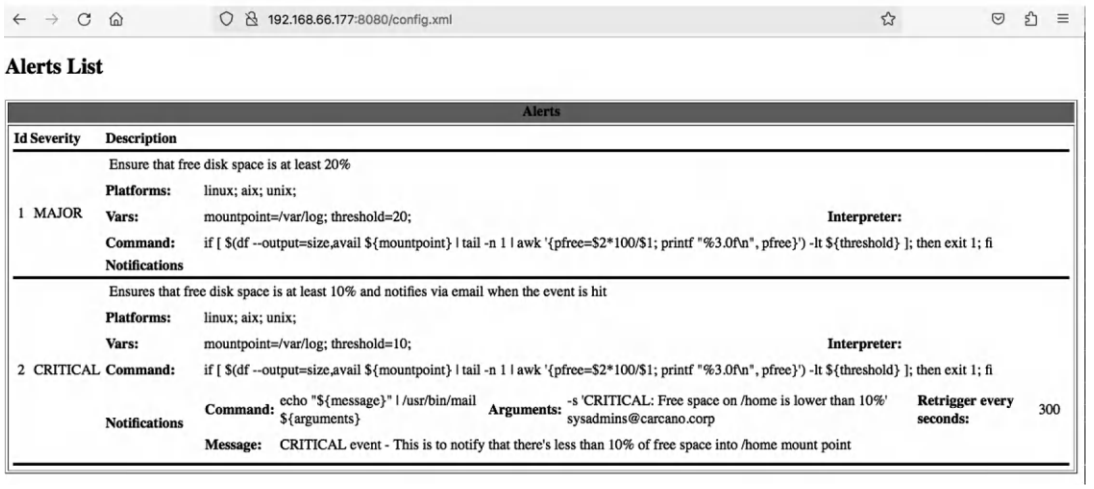
```
python3 -m http.server 8080
```

With our favorite web browser, we can navigate to the URL of the Python web server we just launched: if the Python HTTP server is running on the same computer where we are using the browser, then the URL to open is `http://127.0.0.1:8080/config.xml`. Conversely, we must adjust the IP address to the one of the server running the Python HTTP process.

If you are using the `Vagrantfile` provided with the book, the IP address to use is the one of the `eth1` interface. You easily get it by running

```
ip -4 a show eth1
```

The following picture shows the outcome of the XML to HTML rendering: since the browser detected that the `config.xml` file provides a link to the `xslt/config.xsl` XSLT file, it automatically retrieved the XSLT file and used it to perform the HTML rendering on the fly. This is also revealed by the information logged in the standard output of the Python HTTP server we just launched, which shows the browser requested to download also the `xslt/config.xsl` file.



Once done, terminate the Python HTTP server we just launched by pressing `CTRL+C`.

Note Since we modified some of the Fancymonitor project files, before going on with the rest of the book, we must restore them to their original state: just run `git reset --hard`.

Fancymonitor with XML support

Now that we have a good understanding of the XML format and practiced with command-line tools that can be used as bricks inside shell scripts, it has come the time to see how to process XML in Python.

To go through this topic, we will explore the very first version of the Fancymonitor application, which implements the initial part of a didactical but not trivial Python package which loads its configuration from the XML file we discussed so far.

Change to the `~/devsecops-pillars/fancymonitor/src` directory:

```
cd ~/devsecops-pillars/fancymonitor/src
```

To begin exploring the example, switch to the `f-xml-config` branch which contains the initial implementation:

```
git checkout f-xml-config
```

Note The following examples require the `lxml` Python package. Please install it by running `sudo dnf install -y python3-lxml`.

This initial implementation provides the `carcano.fancymonitor` Python 3 executable package.

The sources are structured in a way not only allowing easy packaging and deployment of the application, but also fostering a structured and tidy development splitting and grouping code blocks in a logical and purposeful way. This eases the overall maintenance process.

Note A Python package is a collection of Python modules organized in a directory hierarchy that includes a special `__init__.py` file. A Python executable package extends a regular Python package by adding a `__main__.py` file, which contains the implementation of the `main` function. This allows the package to be run as a script by invoking Python with the `-m` command-line option followed by the package name.

Table 11-5 summarizes the `src` directory tree structure, highlighting the purpose of each single directory.

Table 11-5. *Fancymonitor's directories*

Directory	Purpose
etc	Store configuration files.
etc/schemas	Store XML schema files.
etc/xslt	Store XSLT files.
log	Store log files.
carcano	The namespace directory tree for the Python libraries developed by the `carcano` vendor. Using a namespace mitigates the risk of naming clashes with a package having the same name developed by other vendors.
carcano/fancymonitor	The directory tree which contains all the Fancymonitor's Python package's source files.
carcano/fancymonitor/models	Subdirectory used for grouping Fancymonitor's source files containing the definitions of objects related to data models. The objects are `Config`, `Probe`, `Statement`, `Alert`, and `Notify`.
carcano/fancymonitor/core	Subdirectory used for storing the Fancymonitor's source files containing the generic core functions, such as the `load_config`. These generic functions will then call specific functions by use case. For example, the `load_config` function calls the XML-specific `xmlloader` to process XML files.
carcano/fancymonitor/loaders	Subdirectory used for grouping the Fancymonitor's source files containing the definitions of objects related to implementation-specific loader functions. For example, the `xml.py` contains the implementation of XML loaders.

The project has been structured in a clean and tidy way, to promote its maintainability – in the next chapters, we'll see how easily it can be extended to support JSON (JavaScript Object Notation) and YAML configuration files.

From the software architecture perspective, each element (the tag nodes) has been implemented as a separate object (a Python class), storing each object into a dedicated Python file in the `models` directory.

These objects are instantiated from functions inside the `xml.py` file stored in the `loaders` directory.

Note Mind how functions guess the path to the XSD file to be used for validation directly from inside the `xml.py` file, using it to perform the actual validation. Validating using the XSD is not only the best practice for schema enforcing when using XML: it also spares us from writing the code for implementing input value checks in the object's setter methods.

The logic for loading the configuration file is inside the `configuration.py` file in the `core` directory.

Note Open each single Python file and check the code inside it: it's mostly self-explanatory. To get the max from this book, I recommend you spend some time reading and understanding these sources.

Before running Fancymonitor, note that the XML configuration file uses a relative path to its XSD schema. This approach has pros and cons: it simplifies deployment by allowing the Fancymonitor project root to be placed anywhere in the filesystem, but it requires launching the application from a directory where the relative path (`../etc`) can be resolved. Since the etc directory is a subdirectory of `src`, you must run it from within `src` (or a subdirectory thereof).

To get started, navigate to the `etc` directory as follows:

```
cd etc
```

At this development stage, you also need to set the `PYTHONPATH` environment variable. This is because `carcano.fancymonitor` is a Python package that Python expects to find in its standard paths.

Normally, packages are installed via `pip` (system-wide, as a user package, or in a virtual environment), but during active development with frequent code changes, it's easier to avoid repeated installations by pointing `PYTHONPATH` to the package's directory.

Let's launch the Fancymonitor executable Python package by running

```
PYTHONPATH=.. python -m carcano.fancymonitor
```

The output will be a dump of the configuration loaded from the XML file into memory. Although it may resemble JSON, it's not formatted as JSON.

Summary

In this chapter, we explored the XML data format in depth, covering key concepts like Document Type Definitions (DTD) and XML Schema Definitions (XSD). We also mastered XPath syntax and gained hands-on experience with command-line utilities such as `xpath` and `xmlstarlet`.

Through the Fancymonitor application example, we demonstrated practical applications, including using XSLT for transforming XML configurations into HTML and examining the structure of a Python executable package that integrates XML support via the `lxml` library.

For DevOps and DevSecOps professionals, expertise in XML – along with related technologies like XSD and XSLT – is essential for effective data interchange, configuration management, and automation. These skills enable efficient data extraction, manipulation, and seamless communication between disparate systems, ultimately fostering operational efficiency and the development of scalable, maintainable solutions in complex environments.

CHAPTER 12

JSON with JQ and Python

Proficiency in JSON (JavaScript Object Notation) is a cornerstone skill for DevOps professionals. As a widely adopted data format, JSON powers configuration files, APIs (such as REST), data interchange in modern applications, and contemporary log aggregators.

Mastering JSON creation and management ensures configurations are structured and readable, which is vital for upholding security and compliance. Skills in querying JSON data using tools like `jq` enable efficient extraction and manipulation of specific information, supporting automation and integration within CI/CD pipelines. These capabilities also bolster security measures, such as extracting insights from logs, validating JSON to prevent injection attacks, and sanitizing inputs to maintain application integrity.

This chapter delves into these concepts comprehensively, beginning with an overview of JSON, its objectives, and its structure. It then explains how to leverage the `jq` command-line tool for integrating JSON support into shell scripts. Finally, it provides sample code for processing JSON in Python, equipping you with practical tools for real-world DevOps scenarios.

Note Before going any further, change to this **chapter's workspace directory** by running `cd ~/devsecops-pillars/ch12`.

What Is JSON

JavaScript Object Notation is an easy-to-parse and lightweight language-independent data-interchange format for the serialization and exchange of structured data.

Its syntax is a subset of the Standard ECMA-404 and defines a small set of formatting rules for the portable representation of structured data. The official JSON website is https://www.json.org.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_12

A JSON document consists of a dictionary of name-value pairs separated using commas: values can be either scalars (number or strings) or complex data types such as lists or nested dictionaries. A JSON document can be easily mapped to data structures used by most of the programming languages, allowing JSON to be easily integrated with any language.

Note In spite of the `J` letter in its name, `JSON` is completely language agnostic and has nothing to do with Java or JavaScript.

To make learning JSON syntax easier, we'll explore it step by step with a hands-on approach using the `jq` command-line utility.

Editing JSON with VIM

The ones of you "ancient" like me probably have a preference for the old-fashioned `vim`.

If you want to use it, install the `vim-enhanced` package by running

```
sudo dnf install -y vim-enhanced
```

Vim provides additional functionalities using plugins: the easiest way for installing them is using the `vim-plug` plugin manager.

Let's install it as follows:

```
curl -fLo ~/.vim/autoload/plug.vim --create-dirs https://raw.
githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

When dealing with JSON, it is convenient to install the `indentLine` plugin: this plugin displays thin vertical lines at each indentation level for code indented with spaces. This feature is very handy to quickly make sense of indentations, especially when dealing with large objects.

To install the `indentLine` plugin using the `vim-plug` plugin manager, you must first modify our user's `~/.vimrc` file (or create it as new if you don't have it) adding the `indentLine` plugin's URL as shown by the following snippet:

```
call plug#begin()
Plug 'https://github.com/Yggdroot/indentLine'
call plug#end()
```

Then, to perform the actual install, launch `vim` as usual:

```
vim
```

Ensure you are in command mode (or press the `ESC` key to switch to it) and type the following command:

```
:PlugInstall
```

You can then verify the status of the plugins as follows:

```
:PlugStatus
```

You must now proceed with configuring the `indentLine` plugin.

Just add the following snippet at the bottom of your `~/.vimrc` file:

```
au! BufNewFile,BufReadPost *.{json} set filetype=json
autocmd FileType json setlocal ft=javascript ts=2 sts=2 sw=2 expandtab
let g:indentLine_char = '|'
let g:indentLine_color_term = 239
```

The above snippet defines the `json` file type and binds the `.json` files to it and then configures the indentation options.

By now, each time you open a `.json` file, vim will use the plugin to visually show the indentation.

JQ – An Overview

JQ is a command-line utility for parsing, filtering, and transforming JSON data. It enables users to extract specific data, manipulate JSON structures, and format output in a flexible and powerful way using a simple query language.

When dealing with shell scripts, it is an invaluable brick to extract values from JSON files.

We can easily install it as follows:

```
sudo dnf -y install jq
```

Its most basic usage is simply parsing and pretty-printing a JSON object – for example

```
echo '{ "number": 73, "name": "Sheldon'\''s number" }' | jq
```

pretty-prints the output as follows:

```
{
  "number": 73,
  "name": "Sheldon's number"
}
```

As you can see, `jq`'s output is colored, multiline, and properly indented: this makes it much more human readable than displaying it in the compact format.

Note `jq` is a handy tool also when you are working with the old-fashioned `vim`: if you are dealing with a plain JSON and you want to pretty-print it in `vim`, simply switch to command mode and type `:%!jq`. Of course, you can apply this trick to whatever you want, for example, with Python `:%!python -m json.tool`.

The most interesting feature provided by `jq` however is manipulating JSON: this feature leverages filters, as depicted by Table 12-1.

Table 12-1. *Filters*

Filter	Description
.Identifier	The identifier of the field you want to get: in the previous example, if you want to get the value of the `number` field, the filter is `.number`. You can even specify only the dot (`.`) as filter to tell `jq` that you want to get the whole input. If you need to return more identifiers, you can list them separated by commas. For example: `'.number, .name'`.
.Identifier?	The same as above, but does not raise an error if the identifier is missing in the input. For example: `'.number, .name, .non_existent?'`.
[i:j]	Returns a slice of the input list, where `i` is the index of the first item of the slice and `j` is the index of the item after the one you want to terminate in the slice. When they are missing, `i` defaults to `0` (the index of the first item) and `j` to the index of the last item.

JQ also provides functions: Table 12-2 summarizes just the most likely to be used ones.

Table 12-2. *Functions*

Function	Description
tonumber	Converts the input into a number
tostring	Converts the input into a string
length	Returns the length (number of elements) of the object
min	Returns the minimum value
max	Returns the maximum value
unique	Removes duplicate items from the list of entities

Filters and functions can be combined in a fully meshed way separating them with a pipe (`|`).

Let us see a few examples – consider the following JSON:

```
{
    "status": "CRITICAL",
    "message": "Free disk space on /home mount-point is critical",
    "threshold": "10%",
    "current": "8%"
}
```

We can easily get the value of the `current` attribute by specifying the `.current` filter:

```
RESULT='{"status": "CRITICAL", "message": "Free disk space on /home mount-
point is critical", "threshold": "10%", "current": "8%"}'
echo "${RESULT}" | jq '.current '
```

The output is

```
"8%"
```

If we want to print it along with a more descriptive message, we can specify it as a string, embedding the filters into it:

```
echo "${RESULT}" | jq -r '"Current consumed percentage is \(.current) while
threshold is \(.threshold)"'
```

The outcome is

```
Current consumed percentage is 8% while threshold is 10%
```

Please note how, since this time we specified the raw option (`-r`), the output is now not enclosed between double quotes.

When dealing with shell scripts, we can exploit the above feature to export the returned values into shell variables. For example, the string to use to export the values as `CURRENT` and `THRESHOLD` is

```
CURRENT="\(.current) THRESHOLD=\(.threshold)"
```

so the whole shell statement to run is

```
export $(echo ${RESULT}|jq -r '"CURRENT=\(.current) THRESHOLD=\
(.threshold)"')
```

Let's check if the variables are correctly set:

```
echo "Current=${CURRENT}, Threshold=${THRESHOLD}"
```

The outcome is

```
Current=8%, Threshold=10%
```

Of course I'm not fostering to always write shell scripts to manage JSON: depending on the use case you may need to use scripting languages natively supporting it, such as Python.

My two cents are: use `jq` when it comes to modify existing scripts to let them parse JSON documents without having to re-code the whole script, saving time and money, or when the operations you have to do on the JSON object are not so complex to require a programming language such as Python. Of course this is just my opinion: there are different opinions out there, which I respect.

Note In the rest of this chapter, I use `jq` to show you some things, but a thorough explanation of its usage would require too much space, and so it is outside of the scope of this book: if you want to learn more, you can find the official documentation at https://stedolan.github.io/jq/manual.

JSON Serialization

Working with JSON documents requires having a good understanding of the JSON syntax.

The most basic JSON document consists of a dictionary (also known as an object) of name-value pairs, where each pair is separated by commas.

The values in these pairs can be either scalar types (such as strings, numbers, or booleans) or complex data types, including lists (arrays) or nested dictionaries (objects).

Comments

The JSON format does not provide a syntax for comments: although you can define your marker for starting comments, this can certainly work only on your application, with no guarantees other applications too will use it as a marker for comments as well.

Null Value

As claimed by one of the changes published by RFC 7159 (it updates RFC 4627), a JSON text is a serialized value.

This means that `null` has officially become a valid JSON literal.

The syntax for specifying `null` is depicted by the following JSON snippet:

```
{ "parent": null }
```

`jq` is able to recognize `null`. We can easily verify this by passing the JSON dictionary to the `map` function and then to the `type` function as follows:

```
echo '{"parent": null}' \ | jq 'map(type)'
```

As expected, the outcome is

```
[ "null" ]
```

Note Be wary that not all JSON parsers/deserializers do support parsing the `null` literal: the risk is to get them raising an invalid syntax exception.

Scalars

Scalars are the most basic formats supported by JSON: they are just booleans, numbers, and strings.

Booleans

As we already said, the RFC 7159 (it updates RFC 4627) claims that a JSON text is a serialized value.

So, same way as `null`, also `true` and `false` have officially become valid JSON literals.

The syntax for specifying booleans is depicted by the following JSON snippet:

```
{ "enabled": true, "autoload": false }
```

`jq` is able to recognize the above literals.

As we already did for null values, we can check this by passing the JSON dictionary to the `map` function and then to the `type` function as follows:

```
echo '{"enabled": true, "autoload": false}' \ | jq 'map(type)'
```

As expected, the outcome is

```
[ "boolean", "boolean" ]
```

Numbers

Numbers can be either integers or double-precision floating-point. The actual syntax is depicted by the following JSON snippet:

```
{ "integer": 42, "floating-point": 1.56, "other-floating": 0.9,
"exponent": 10E-2 }
```

Note Valid exponent character markers are `e`, `e+`, `e-`, `E`, `E+`, and `E-`.

Let's check the types inferred from `jq`:

```
echo '{"integer": 42, "floating-point": 1.56, "other-floating": 0.9,
"exponent": 10E-2}' | jq 'map(type)'
```

As expected, the outcome is

```
[ "number", "number", "number", "number" ]
```

Strings

Strings are declared by enclosing the value with double quotes as shown by the following JSON snippet:

```
{ "FullName": "Marco Antonio Carcano" }
```

Let's check the type inferred from `jq`:

```
echo '{"FullName": "Marco Antonio Carcano"}' | jq 'map(type)'
```

As expected, the outcome is

```
[ "string" ]
```

Be wary also that according to the specification:

> *A string is a sequence of Unicode code points wrapped with quotation marks (U+0022). All characters may be placed within the quotation marks except for the characters that must be escaped: quotation mark (U+0022), reverse solidus (U+005C), and the control characters U+0000 to U+001F. There are two-character escape sequence representations of some characters.*

For the sake of clarity, some of the involved Unicode control characters from `U+0000` to `U+001F` are

- `U+0000` <control-0000> (NUL: NULL) (used in null-terminated strings)
- `U+0009` <control-0009> (HT: HORIZONTAL TABULATION) (inserted by the tab key)
- `U+000A` <control-000A> (LF: LINE FEED) (used as a line break)
- `U+000C` <control-000C> (FF: FORM FEED) (denotes a page break in a plain text file)
- `U+000D` <control-000D> (CR: CARRIAGE RETURN) (used in some line-breaking conventions)

The consequence of this claim is that you cannot insert into a string any of these control characters, neither the quotation mark (`U+0022`) nor the reverse solidus (`U+005C`), without escaping them first.

Multiline Strings

JSON serializers/deserializers do not have a standard way to handle multiline strings, so it's up to the application to decide how to deal with them.

There are two possible approaches:

If your serializer/deserializer manages the escaping of `\n` (`U+000A`) control character as a newline, you can try the following syntax:

```
{ "Multiline-string": "first line\\nSecond line" }
```

Otherwise you can try the dirty workaround of splitting the multiline string into a list of single-line strings (we will discuss lists shortly, when talking about complex data types).

In this case it is up to your application to treat them as a single multiline string:

```
{ "Multiline-string": [ "first line", "Second line" ] }
```

Let's check the type guessed from `jq`:

```
echo '{"Multiline-string": [ "first line", "Second line" ]}' \
| jq 'map(type)'
```

This time the outcome is

```
[ "array" ]
```

In this context, an array is actually a list – it is often called like so indeed.

Complex Data Types

As we said, JSON also supports complex data types, such as dictionaries and lists.

Objects (Dictionaries)

Besides the empty JSON document `{}`, the most basic JSON document looks as follows:

```
{ "number": 73 }
```

The above JSON contains just a scalar key called `number` with the value 73. When using `jq`, access to the item's value just requires prepending the item's key with a dot. For example:

```
echo '{ "number": 73 }' | jq -r '.number'
```

An example of a key with complex data is

```
"details": { "os-family": "Linux", "family": "redhat" }
```

We can add the `details` item to the previous JSON document: the outcome is a JSON object with a nested dictionary.

The outcome is shown in the `os.json` file in this chapter's workspace directory:

```
{
  "number": 73,
  "details": {
    "os-family":"Linux",
    "family": "redhat"
  }
}
```

Let's use it to run a few query examples helping to better understand how to operate with `jq`.

The most trivial need is getting the whole `details` dictionary:

```
jq '.details' os.json
```

A less straightforward yet quite common need is to list just the keys of the JSON dictionary.

This is achieved by piping the output of the filter to the `keys` function as follows:

```
jq '.details|keys' os.json
```

The output is as follows:

```
[ "family", "os-family" ]
```

If we want to get a specific item of the details dictionary, we must provide the whole path to it:

```
jq -r '.details.family' os.json
```

A common caveat is dealing with keys having the minus (-) character inside. The trick for overcoming it is to enclose the key by double quotes. For example, to get the `os-family` item of the `details` dictionary

```
jq -r '.details."os-family"' os.json
```

The output indeed is

```
Linux
```

Note When dealing with keys with the - character in their name, it is necessary to enclose in double quotes because omitting them causes `jq` to evaluate the key as a subtraction between the tokens split by the - character.

Arrays

This data structure is a list of values.

Note To highlight the affinity between JSON and YAML, since YAML calls list (or even sequence) the lists of values, from this point onward we use the word list in place of array, without worrying about the storage differences and underlying mechanisms for seeking the list items.

The following example depicts the JSON syntax of a list:

```
[ "i686", "x86_64", "aarch64" ]
```

To experiment a bit with a few `jq` commands regarding managing lists, let's assign the above list to the `JSON_LIST` variable:

```
JSON_LIST='[ "i686", "x86_64", "aarch64" ]'
```

Working with Arrays

So far, we have learned the syntax of a JSON document and how to perform basic queries on dictionary items using `jq`.

However, real-life use cases are often much more complex, involving JSON documents with nested dictionaries and lists.

In this section, we will focus on how to effectively handle lists and write queries for the most common use cases.

Number of Elements Using the Length Filter

A common need is certainly getting the number of elements in the list.

This can be addressed by piping the output of the filter to the `length` function:

```
echo "${JSON_LIST}" | jq '. | length'
```

Getting One Element of the List

Elements of a list are accessed by specifying their index, we can use `jq` filters to retrieve a list item by supplying its index.

For example, since list index is 0 based, we can access the third element of the list as follows:

```
echo "${JSON_LIST}" | jq -r .[2]
```

The output is as follows:

```
aarch64
```

Getting a Slice of the List

Another common need can be getting a slice (so a subset) of a list.

For example, the `jq` statement to get items from the second to the last of the list is

```
echo "${JSON_LIST}" | jq .[1:]
```

As we previously saw, `[i:j]` returns a slice of the input list, where `i` is the index of the first item of the slice and `j` is the index of the item after the one you want to terminate in the slice. When they are missing, `i` defaults to 0 (the index of the first item) and `j` to the index of the last item.

Sorting a List

Sometimes it is necessary to sort a list. This can be achieved by piping to the `sort` function.

For example:

```
echo "${JSON_LIST}" | jq '.| sort'
```

It of course works also with a list of numbers:

```
NUMBERS='[ 1, 0.3, 2.5, -0.9 ]'
echo "${NUMBERS}" | jq '.| sort'
```

The output is as follows:

```
[ -0.9, 0.3, 1, 2.5 ]
```

Getting Min and Max

If needed, we can pipe the output of the filter to the `min` function to get the minimum element:

```
echo "${NUMBERS}" | jq '.| min'
```

It returns `-0.9`.

Conversely, if we need the maximum element, we can pipe the output of the filter to the `max` function:

```
echo "${NUMBERS}" | jq '.| max'
```

It returns `2.5`.

Unpacking a List

If necessary, you can unpack a list, splitting its items one per line. For example:

```
echo "${NUMBERS}" | jq -r '.[]'
```

The outcome is

```
1
0.3
2.5
-0.9
```

As you can see this is no longer a JSON object – the square brackets and the commas are gone: it is just text on multiple rows now.

The consequence is that a statement like this one is not valid:

```
echo "${NUMBERS}" | jq -r '.[] | sort'
```

since it relies on the `jq`'s built-in `sort` function that expects a JSON array as input.

Since in this specific case `jq` generated a list of elements with one element per row, if we really want to sort the items, we must instead rely on the `sort` shell's command-line utility:

```
echo "${NUMBERS}" | jq -r '.[]' | sort -n
```

In certain scenarios, you may need to transform each item of a JSON array into a separate JSON object (or dictionary).

Each object will contain a key field that represents the index of the item in the array and a value field that holds the actual value of the item.

For example:

```
echo "${JSON_LIST}" | jq '.| to_entries[]'
```

processes the JSON array stored in the `JSON_LIST` variable and exploits the `to_entries` function to convert each item into an object with `key` and `value` fields.

The output is

```
{ "key": 0, "value": "i686"}
{ "key": 1, "value": "x86_64" }
{ "key": 2, "value": "aarch64" }
```

An example use case is to pipe the output of `to_entries[]` to the `select` function to filter for a specific value, as demonstrated in the next paragraph.

Getting the Index of an Item of the List with a Certain Value

By combining the `to_entries[]` and `select` functions, we can easily retrieve the index of the list item with a specific value.

For example, to get the dictionary item having the `x86_64` value:

```
echo "${JSON_LIST}" \
| jq '.| to_entries[] | select(.value=="x86_64").key'
```

The output is `1`.

Checking the Existence of a Specific Value

We can of course check if a list contains a specific value. This can be achieved by using the `select` and `index` functions.

For example:

```
echo "${JSON_LIST}" | jq -r '.[]|select( index("x86"))'
```

If the argument passed to the `index` function is not among the list's items, then it returns nothing.

More Complex Use Cases

We are now prepared to explore more complex use cases. For instance, the `distros.json` file located in the workspace directory of this chapter contains a JSON document that includes a list of dictionaries, each representing a different Linux distribution.

Each single list item has the following format:

```
{
    "name": "alpine",
    "number": 73,
    "major": "3",
    "details": {
        "OS-Family": "Linux",
        "distro-family": null
```

```
    },
    "available": false,
    "releases": [
        "3.0",
        "3.1"
    ],
    "arch": [
        "x86",
        "x86_64"
    ]
}
```

To make the example more challenging, not every list item has the `available` key.

Key as Matching Criteria

A very common need when working with complex JSON documents is extracting list items having a key matching specific criteria.

We can achieve that by using the `map` and the `select` functions together.

The below statement selects only the list items having `.details.distro-family` matching the string `redhat`:

```
jq -r 'map( select( .details."distro-family"=="redhat" ) )' distros.json
```

List Item as Matching Criteria

Another common requirement is to extract elements from a list where a specific key contains a list of values, and we want to check for the presence of a particular element.

We can accomplish this by using the `map` and `select` functions together, passing the key to be processed to the `index` function to verify if it contains the desired element.

As an example, the following statement selects only the items from the list that have a value containing the word `x86` within the arch key's values:

```
jq 'map( select( .arch | index("x86")))' distros.json
```

Boolean Operators Inside the Select Function

In real life, conditions are often more complex and involve the use of boolean operators to specify multiple matching criteria.

An example use case is selecting only the list items having `.details.distro-family` matching the string `redhat` and number lower than `78`:

```
jq 'map(select(.details."distro-family" == "redhat" and .number<78))'
distros.json
```

In the following statement, we see an example of using boolean operators with a `select` condition having both a key with a matching value and a key having a matching element among its list of items:

```
jq 'map(select(.details."distro-family" == "redhat" and (.arch |
index("x86_64"))))' distros.json
```

Select Items Containing a Specific Key

Sometimes you may have to deal with lists of dictionaries having optional items. In such a scenario, it is quite common having to filter items relying on these optional items.

As an example, you may need to check which items have a specific key.

This can be achieved using the `has` function as follows:

```
jq -r 'map(has("available"))' distros.json
```

The outcome is

```
[ false, false, true ]
```

So the outcome is just a map of booleans highlighting the index of the matching items.

To get the actual items, we can combine it with the `select` function as follows:

```
jq -r 'map( select( has("available")))' distros.json
```

Processing JSON with Python

In the previous chapter we had a look at the initial implementation of the Fancymonitor application, which supports only XML.

After having familiarized with JSON, we can now have a look at the feature branch providing JSON support to it.

First, change to the Fancymonitor project root directory as follows:

```
cd ~/devsecops-pillars/fancymonitor
```

and check out the `f-json-config` branch:

```
git checkout f-json-config
```

The changes made to add the feature for supporting JSON configuration files can be easily shown by comparing the current branch (`f-json-config`) to the previous one (`f-xml-config`).

The first interesting information is the list of modified files – we can get it as follows:

```
git diff --name-only f-xml-config
```

We can now use `git diff` to show the specific changes to each single file.

For example, to see the changes made to the `src/carcano/fancymonitor/models/alert.py` file, run

```
git diff f-xml-config -- \
src/carcano/fancymonitor/models/alert.py
```

Before going on with the read, please check the changes done at each single file.

As you can see, the changes made to the files in the `models` directory always follow the same pattern:

- Import the Python's `json` library.
- Implement the `__getattribute__` method.
- Implement a class derived from `json.JSONEncoder` that can be exploited as necessary to serialize the object into JSON format using the `json.dumps` function.

Since JSON, conversely from XML, is schemaless, it was also needed to add some checks to verify the input values. For this reason, it has also added a validation for

- The `severity` attribute of the `Alert` class
- The `trigger_every` attribute of the `Notify` class

As for the other files, in the `loaders` directory, the new file `json.py` was added: it contains the logic for loading the JSON configuration and instantiating the objects.

It was of course necessary to modify the `configuration.py` file in the `core` directory to try to parse and load the configuration as a JSON document if parsing it as an XML should fail.

Last but not least, the `main` function in the `__main__.py` file was modified to dump the configuration object as a JSON using the `json.dumps` function.

In the `etc` directory, it was obviously added also the `config.json` file with the JSON configuration.

Note Despite being unrelated to loading from the JSON format, the `Defaults` object was also created: this will be used to provide values of not specifically specified settings. We will see it in action when dealing with the YAML format.

In the same way as before, to run it we must be in the `src/etc` directory and set the `PYTHONPATH` environment variable as follows:

```
cd src/etc
PYTHONPATH=.. python -m carcano.fancymonitor -c config.json
```

To avoid loading the default configuration file, we used the `-c` command-line option to provide the path to the JSON configuration file.

This time it dumped the configuration in JSON format – if you fancy, you can even pipe it to `jq`.

Note Mind how it can still process the `config.xml` file, converting its contents into JSON: just run `PYTHONPATH=.. python -m carcano.fancymonitor`.

I hope you can see the value of being wise, spending some time designing and structuring things in a tidy, provident, and clean way: thanks to this, the improvement step to add JSON support was an easy walk and didn't require any rework – we just added new files and some additional code to the already existent ones, but we didn't need to delete or rework anything.

Summary

In this chapter, we explored JSON (JavaScript Object Notation), focusing on its structure and processing with `jq`. Through explanations and examples, we mastered querying JSON data using `jq` syntax, addressing common use cases. This empowers us to confidently create JSON documents and craft complex `jq` queries for extracting specific data subsets.

These skills enable seamless integration of `jq` into scripts for CI/CD pipelines or REST (Representational State Transfer) API interactions. Additionally, by examining the JSON-related branch of the Fancymonitor project, we learned to process JSON documents programmatically in Python.

For DevOps and DevSecOps professionals, JSON proficiency is essential for data interchange, manipulation, and modern software development. It's ubiquitous in APIs, configuration files, and data storage, enabling smooth interaction with web services and applications. Expertise in `jq` supports efficient extraction, transformation, and analysis of information, automating tasks, manipulating data streams, and generating reports to enhance operational efficiency.

Together, these JSON and `jq` skills equip practitioners to streamline workflows, improve data handling, and ensure robust communication between systems, ultimately fostering more agile and reliable software delivery processes.

CHAPTER 13

YAML with Yq and Python

Understanding YAML and its advanced features is essential for DevOps professionals, as it's a cornerstone format for configuration files across tools and platforms, including CI/CD pipelines, infrastructure as code (IaC), and application deployments. Mastering YAML semantics, tags, anchors, aliases, and the merge key enables the creation of efficient, maintainable, and reusable configurations that adhere to the DRY (Don't Repeat Yourself) principle.

This chapter delves deeply into this critical topic, beginning with an exploration of the YAML format. We'll learn to use `yamllint` for syntax validation, minimizing deployment failures in YAML-based configurations. Additionally, we'll cover `yq` for enhancing automation in shell scripts, streamlining workflows. Finally, we'll demonstrate processing YAML with tags using Python and the `PyYAML` library, facilitating seamless integration of configuration management into Python applications.

These skills collectively enhance the security and efficiency of DevOps and DevSecOps practices, empowering professionals to build robust, error-free, and scalable infrastructure solutions.

Note Before going any further, change to this **chapter's workspace directory** by running `cd ~/devsecops-pillars/ch13`.

What Is YAML

YAML is a Unicode-based, human-readable data serialization language designed for configuration files and manifests.

Its compact and readable syntax, combined with DRY features like anchors and aliases, makes it highly efficient for structured data representation.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_13

A key advantage of YAML is its support for document streams – multiple YAML documents separated by `---` within a single file. This allows parsers to process individual chunks sequentially without loading the entire file into memory, enabling efficient handling of large datasets.

Additionally, YAML's explicit typing and support for complex data structures (e.g., lists, maps, and nested objects) make it versatile for both simple and advanced use cases.

To have a look at existing YAML files, since Ansible heavily relies on the YAML format, we can install the `rhel-system-roles` RPM package. This package provides Red Hat's official Ansible roles, containing a lot of YAML-formatted files we can use for exploring this format.

Just run:

```
sudo dnf install -y rhel-system-roles
```

These roles are installed beneath the `/usr/share/ansible/roles` directory: we will use them in some of the examples in this section.

Editing YAML with VIM

If, like me, you are addicted to the old-fashioned `vim`, you could not avoid using the `vim-yaml` and `indentLine` plugins.

Since we have already downloaded the `indentLine` plugin in the previous chapter using the `vim-plug` plugin manager, we can do the same for downloading `vim-yaml`.

Modify the `~/.vimrc` file adding the URL of the `vim-yaml` plugin.

Once done it must look like the following snippet:

```
call plug#begin()
Plug 'https://github.com/Yggdroot/indentLine'
Plug 'https://github.com/stephpy/vim-yaml'
call plug#end()
```

Then, to do the actual install, launch vim as usual:

```
vim
```

As we already did when installing the `indentLine` plugin, ensure you are in command mode (or press the `ESC` key to switch to it) and type the following command:

```
:PlugInstall
```

We must now proceed with configuring the `vim-yaml` plugin.

Just add the following snippet at the bottom of your `~/.vimrc` file:

```
au! BufNewFile,BufReadPost *.{yaml,yml} set filetype=yaml
autocmd FileType yaml setlocal ts=2 sts=2 sw=2 expandtab
```

The above directives define the `yaml` file type, bind `.yaml` and `.yml` files to it, and configure the YAML indentation.

Note If you want to alter the suggested configuration settings, be wary that YAML does not allow tabs, and the suggested indents are multiples of two spaces.

As for the `indentLine` plugin, we leave the settings we made in the JSON chapter unchanged.

To see this plugin in action, we can now open one of the files provided by the Ansible role, such as `/usr/share/ansible/roles/rhel-system-roles.timesync/meta/main.yml`.

When editing large YAML files, you may find it useful switching to the `indent` fold method: in command mode, just type `:set foldmethod=indent`. Once switched, the whole document is shown collapsed and you can use arrows to navigate across the folded nodes. You can then expand specific nodes using the right arrow key or by typing `zo` or collapse by typing `zc`. When you reach the part of the document you want to change, you can edit it as you would normally do, even within folded sections. To switch back to the regular view, in command mode, just type `:set foldmethod=expr` or `:set foldmethod=syntax`.

Yq – An Overview

Note Be wary that there are two different projects called `yq`: https://github.com/mikefarah/yq and https://github.com/kislyuk/yq. In this book, we talk about the latter: it seems to be the more broadly used, indeed probably because the first one is not 100% `jq` compatible.

Yq is a handy command-line tool for managing YAML documents in shell scripts, often described as "the `jq` for YAML." It acts as a wrapper around `jq`, allowing users to convert YAML to JSON and manipulate it using `jq`'s powerful querying capabilities.

Mind that the `yq` implementation we are about to install relies on `jq` for processing JSON data.

If you have not installed it already while reading the previous chapter, install it now as follows:

```
sudo dnf -y install jq
```

Since Yq is a Python module, we must first install the Python PIP module:

```
sudo dnf -y install python3-pip
```

Note PIP is a package management tool for Python which allows users to install, upgrade, and manage Python packages from the Python Package Index (PyPI). PyPI is a repository for Python software packages, allowing users to find, install, and share Python libraries and tools.

We can now then invoke `pip` to install `yq` as follows:

```
sudo python3 -m pip install yq
```

Note It is generally not advisable to install a PIP module system-wide due to potential conflicts with other Python packages and dependencies. However, in this case, we are making an exception because `yq` is a well-known command-line utility with a moderate risk impact. Additionally, we want the `yq` tool to be available for all users, eliminating the need to distribute it individually to each user who may run scripts containing `yq` statements.

To see `yq` in action, we can use it to extract the contents of the task `Enable phc2sys` from the `timesync` Ansible role. The timesync role's root directory is located at `/usr/share/ansible/roles/rhel-system-roles.timesync`, and the role's tasks are defined in the `tasks/main.yml` file.

In Ansible, tasks are typically defined as dictionaries within a list. To extract a specific dictionary item, we need to select the one that has the matching name.

Since the `yq` filter uses a syntax similar to that of `jq`, the filter to use is

```
'. | to_entries[] | select(.value.name=="Enable phc2sys") | .value'
```

Let's put everything together and run the following command:

```
yq -y '. | to_entries[] | select(.value.name=="Enable phc2sys") | .value' /
usr/share/ansible/roles/rhel-system-roles.timesync/tasks/main.yml
```

The output is

```
name: Enable phc2sys
service:
  name: phc2sys
  state: started
  enabled: true
when:
  - timesync_mode == 2
  - timesync_mode2_hwts
```

Note For the sake of completeness, a task is the smallest runnable unit in Ansible: it is used to instantiate an Ansible module providing the necessary parameters for running it. The above task instantiates the `service` module passing `name`, `state`, and `enabled` as parameter. The task is run only if the `timesync_mode` variable value is `2` and the `timesync_mode2_hwts` variable is set.

So, the good news with `yq` is that we are not required to learn anything more than we already know about using `jq`. We just need to supply the `-y` command option to get the output converted from JSON back to YAML format, since `yq` by default outputs in JSON format.

Let's see a few more examples: the `tags/custom.yml` from the chapter's workspace directory contains a probe object reworked into YAML format.

We can run a few queries on it such as extracting the `alert` node:

```
yq .alert tags/custom.yml
```

Since we omitted the `-y` command option, this time the output is in JSON format.

We may want to get raw data from a node – in this case just use the `-r` command option as follows:

```
yq -r .statement.command tags/custom.yml
```

YAML Serialization

YAML serialization is quite a huge topic: here we focus only on the things that you are most likely to hit in the daily work.

As we already did in the previous chapter when explaining JSON, to ease learning the YAML syntax, we will go through it step by step, with a hands-on approach relying on the `yq` command-line utility.

Indentation-Based and Flow-Style Notations

YAML has two primary notations: `indentation-based` and `flow-style` notation (sometimes also called `block` and `compact`).

In the `indentation-based` notation, YAML stores information in a structured format using lists and dictionaries, which are organized through indentation and colon-separated key-value pairs.

In the `flow-style notation`, lists are enclosed by square brackets `[...]` whereas dictionaries are enclosed between `{...}`. This notation provides a compact representation that resembles JSON in its structure, making it familiar to those who have experience with JSON.

Note Be wary that YAML keys are case sensitive.

Node Types

In a YAML document, a unit of data representing a piece of information is called a Node. There are only three types of nodes as shown in Table 13-1.

Table 13-1. *Node types*

Type	Description	Example
Scalar	A simple value, such as a string, number, boolean, or null	`id: 1`

Sequence	A list of items – a sequence is roughly the same concept as an array used in programming languages	`memberOf:`
		`  - devops`
		`  - security`

Mapping	A set of key-value pairs, with the restriction that each of the keys is unique (a mapping behaves the same way as dictionaries or associative arrays used in programming languages)	`userdata:`
		`id: 1`
		`username: "mcarcano"`
		`locked: false`

Document Separator

You can have multiple documents within a single file by separating them explicitly marking their beginning using three hyphens (`---`).

Note Please note that there exist also the three dots (`...`) marker in YAML, which indicates the end of a document. When the parser encounters this marker, it stops processing the current document and will not parse any content that follows it. This can be useful for debugging and troubleshooting, as it allows you to clearly delineate the end of a document.

Comments

Comment lines begin with the # character. For example:

```
# This is a YAML comment
foo: bar # This is also a YAML comment
```

null Value

The `null` word identifies the null value. For example:

```
my_undefined: null
```

Note Be wary that YAML specifications also consider both an empty value and the ~ character as a null value.

Scalars

Scalars are the most basic types supported by YAML.

Booleans

Boolean values can be either `true` or `false`. They must not be enclosed in quotes. For example:

```
toggle: true
```

Numbers

Numbers must not be enclosed in quotes. Let's see some examples:

```
fixed: 42.73
exponential: 0.42e+3
sexagesimal: 190:20:30.15
negative infinity: -.inf
not a number: .nan
```

Strings

Strings do not normally require quotation marks, although you are free to enclose them in either double or single quotes.

```
msg: this is a string
msg: 'this is another string'
msg: "this is yet another a string"
msg: "42"
msg: "true"
```

Note however that sometimes quoting the string is mandatory. For example:

- If the string contains a colon (`:`) followed by a space character
- If the string begins with an opening curly brace { or square bracket [, since they are used in the flow-style notation to define list items or maps
- If the string is literal such as `true` or `false`: this is to avoid having the string mapped to a different type, such as boolean.

As a real-world example, the JINJA2 templating engine uses curly braces to access variables and print their value – since Ansible supports the JINJA2 template engine, to avoid confusion with flow-style notation dictionaries, a string starting with a curly brace { must be enclosed by double quotes as follows:

```
msg: "{{ variable }}"
```

Multiline Strings

Depending on your needs, there are two distinct ways to handle multiline strings.

If you need to preserve newlines, you must use the vertical bar (`|`) – for example:

```
preserve_newlines: |
  Carcano AG
  Paradeplatz, 123ABC
  Zurich, 8000
  CH
```

If instead you want to fold newlines (have newlines converted into spaces), use the greater-than (`>`) character – for example:

```
fold_newlines: >
  Marco Antonio
  Carcano
```

Note Although they are not very commonly used, there are two additional ways to deal with multiline strings: the first is to enclose the string between single quotes and go to the next line as necessary. The other is much more of a trick: enter the string on a single line between double quotes adding the new line sequence `\n` in the spot where the newlines are supposed to be inside it.

Dictionary

Dictionaries (AKA mapping in YAML's terms) can be declared using both `indentation-based` and `flow-style` formats.

The following snippet shows how to declare the `vars` dictionary using the `indentation-based` format:

```
vars:
  package_name: "apache"
  service_name: "httpd"
  port: 80
```

The following snippet shows the same dictionary, this time represented in flow-style format:

```
vars: { package_name: "apache", service_name: "httpd", port: 80 }
```

Note To define an empty dictionary, use `{}` as value. For example: `vars: {}`.

List

Lists (AKA sequences in YAML's terms) can be declared using both `indentation-based` and `flow-style` formats.

The following snippet shows how to declare the `platforms` list using the `indentation-based` format:

```
platforms:
  - "linux"
  - "aix"
  - "unix"
```

The following snippet shows the same list, this time represented in flow-style format:

```
platforms: [ "linux", "aix", "unix" ]
```

Note To define an empty list, use `[]` as value. For example: `platforms: []`.

List of Dictionaries

It is quite common to have to deal with lists of dictionaries.

The syntax is as depicted by the following snippet:

```
packages:
  - name: "apache"
    service: "httpd"
    port: 80
  - name: "tomcat"
    service: "tomcat"
    port: 8080
```

Tags

Tags are identifiers starting with an exclamation mark (`!`), put between the key and the value, used to suggest to the parser which class to use to decode the node's contents.

They are typically used when dealing with structured contents, such as list, dictionaries, or even more complex contents including nested list and dictionaries, to suggest to the parser which object to use to represent them in memory and how to load the values into it.

We will see two practical examples of this at the end of this chapter.

Tags can also be used to enforce a specific type for a node, overriding the automatically detected type.

For example:

```
enabled: "true"
```

Because of the double quotes enclosing the value, this node is automatically mapped to a string.

We can instead force it to be mapped to a boolean by prepending the built-in `!!bool` tag as follows:

```
enabled: !!bool "true"
```

There are three types of tags: `Default`, `Custom`, and `Global`.

Default

Default tags are a set of built-in tags that correspond to common data types. These tags are prefixed with an additional exclamation mark (`!`), resulting in a double exclamation mark (`!!`). This convention helps distinguish `Default` tags from `Custom` and `Global` tags.

Table 13-2 summarizes all the available `Default` tags.

Table 13-2. *Default tags*

Default Tag	Bound Data Type
`!!null`	Null
`!!bool`	Boolean value, such as true or false
`!!int`	Integer number, such as 42
`!!float`	Floating-point number, such as 42.73
`!!str`	A string, such as “Marco Antonio Carcano”
`!!seq`	A list
`!!map`	A dictionary

When they are omitted, the YAML parsers automatically guess the node type and map it to any of the above ones.

Custom

These are custom-defined tags such as `!probe`. They are used to suggest to the parser which class to use to load the document section marked with that tag. For example, if we would want to improve the Fancymonitor application by adding YAML support, we would need to add `Custom` tags to the YAML file to suggest to the parser which class to use to load each data section. Table 13-3 lists examples of possible `Custom` tags along with the mapped object.

Table 13-3. *Example custom tags for the Fancymonitor application*

Custom Tag	Bound Class
`!probe`	Probe
`!probe:statement`	Statement
`!probe:alert`	Alert
`!probe:alert:notify`	Notify

The tags/custom.yml file in the chapter's workspace shows an example of how these tags have been added to the YAML contents.

Global

These are `Custom` tags compliant to the IETF RFC 4151 tag URI scheme, which provides a unique identifier for the tag, to avoid naming conflicts and enhance interoperability. Using globally unique tags prevents the risk of having tags with the same name mapped to different mismatching objects, perhaps developed by different vendors.

The example `Custom` tag we defined in the previous paragraph is safe until the Fancymonitor application is available only inside our organization: if the application then becomes publicly available, third parties might develop extensions hitting tag names potentially raising conflicts.

To mitigate this risk, the previously defined `Custom` tags can be turned into `Global` by transforming them as follows:

- `!<tag:carcano.ch,2000:fancymonitor>`
- `!<tag:carcano.ch,2000:fancymonitor:probe>`
- `!<tag:carcano.ch,2000:fancymonitor:probe:statement>`
- `!<tag:carcano.ch,2000:fancymonitor:probe:alert>`
- `!<tag:carcano.ch,2000:fancymonitor:probe:alert:notify>`

Note The best practice is using a globally available top-level domain you really own and use the year you started owning it.

The `tags/globals.yml` file in the chapter's workspace shows an example of how these tags have been added to the YAML contents.

Long story short, if your YAML is consumed by a publicly available application, and you suppose a third party will sooner or later start writing custom libraries around it, your best option is to use `Global` tags.

It is worth mentioning that `Default` tags are just the shorthand of `Global` tags belonging to `yaml.org`. For example:

- `!!str` is the shorthand of `!<tag:yaml.org,2002:str>`
- `!!seq` is the shorthand of `!<tag:yaml.org,2002:seq>`
- `!!map` is the shorthand of `!<tag:yaml.org,2002:map>`

Note We'll see YAML tags in action later on with an example using the `PyYAML` Python package.

Versions and Schemas

The YAML specifications are available on the official website.

The current release at the time of writing this part of the book is 1.2.2, which was published with the claimed purpose of bringing YAML "into compliance with JSON as an official subset."

Conversely from JSON, which has been specifically designed to promote its usage for data exchange between different systems, YAML is instead more oriented at writing manifests, so it is often used with a specific processor.

Since a processor often implements a specific version of YAML, when writing a YAML document, you must pay attention to be compliant with the version implemented by your processor or try to write the YAML code in the most portable way.

Version 1.2, besides adding support to UTF-32, introduces the concept of `Schema`, which defines how tags in a YAML document relate to data types in programming languages.

It provides three different schemas:

- **Failsafe schema**: A minimal schema that allows for the most basic YAML features
- **JSON schema**: A schema that aligns with JSON data types
- **Common schema**: A more comprehensive schema that includes additional types

For example, these are the tags defined in the YAML 1.2 specifications' JSON schema:

- `!!str`
- `!!map`
- `!!seq`
- `!!null` A null value. Matches: `null`
- `!!bool` Boolean. Matches: `true | false`
- `!!int` Integer. Matches: `0 | -? [1-9] [0-9]*`
- `!!float` Float. Matches: `-? ( 0 | [1-9] [0-9]* ) ( \. [0-9]* )? ( [eE] [-+]? [0-9]+ )?`

If you do not have special needs, such as the extra types provided by the core schema, you can simply follow the YAML 1.2 JSON schema: this way your YAML code should work for most YAML 1.1/1.2 processors. Be wary that the 1.1 specifications support only UTF-8 and UTF-16, recommending using UTF-8 as the output encoding.

Note Despite the 1.1 specifications being superseded by the 1.2, many broadly used serialization libraries still implement 1.1.

Everything explained in this chapter refers to the 1.2 JSON schema.

Anchors and Aliases

Anchors and aliases are part of the YAML feature set aimed to support the Don't Repeat Yourself (DRY) approach, which in this context means avoiding repetitions in the document.

- The & character defines an `anchor`: this acts as a marker for a specific chunk of data.
- The * character creates an alias: this is a reference to a previously defined `anchor` that expands to the contents of the anchor it refers to.

The `dry/tomcat.yml` file contains an example snippet of an example Ansible `vars` file. It contains a few anchors and aliases to demonstrate their practical usage.

Let's have a look at it:

```
cat dry/tomcat.yml
```

As you can see:

- At line 3, we define the `&tomcat_opts` anchor with lines 4 and 5 as contents.
- At line 9, we use the `*tomcat_opts` alias to refer to the contents of the `&tomcat_opts` anchor for defining the first tomcat instance's option – simply put, it just copies the node as the value of the `options` node at line 8

When a YAML parser finds anchors and aliases, it processes them, producing the final YAML document with all the expanded contents.

To see it in action, we can parse the `dry/tomcat.yml` file using `yq` as follows:

```
yq -y . dry/tomcat.yml
```

The output is

```
defaults:
  options:
    opts: -Xms1G -Xmx2G
    catalina_home: /opt/tomcat
instances:
  - name: instance-01
    options:
      opts: -Xms1G -Xmx2G
      catalina_home: /opt/tomcat
  - name: instance-02
    options:
      opts: -Xms2G -Xmx2G
      catalina_home: /opt/tomcat
      catalina_base: /opt/tomcat-02
```

So, as expected, the anchors and aliases have been expanded.

The node with the name `instance-01` uses the defaults referenced by the `&tomcat_opts` anchor, whereas the node with the name `instance-02` uses explicitly defined values.

But what happens if we try to add an alias to a node that already has contents? For example, what happens if the options key at line 8 already contains the `catalina_base` key with the value `/opt/tomcat-01` and we try to apply an alias to it? This scenario is depicted by the following snippet (modify the contents of the `dry/tomcat.yml` to have the `instance-01` key matching this snippet):

```
instances:
  - name: instance-01
    options:
      *tomcat_opts
      catalina_base: /opt/tomcat-01
```

then parse it with `yq`:

```
yq -y . dry/tomcat.yml
```

It seems this time we are going nowhere:

```
yq: Error running jq: ParserError: while parsing a block mapping
  in "dry/tomcat.yml", line 7, column 5
did not find expected key
  in "dry/tomcat.yml", line 10, column 7.
```

This error is raised because we attempted to expand an alias inside a node having already a value. This is not supported, since the expansion would replace the whole contents.

This is clearly a merging operation, so to address it we need to use the `merge key` feature.

Merge Key

The `Merge Key` is a special key represented by the `<<` symbol that enables you to include the contents (i.e., merging) of an aliased node into another node.

Since the contents are expanded within the merge key, there is no problem if the node already contains data: even better, if the node already has keys that match those in the aliased content, the values from the aliased content get overridden by the existing local values.

This override capability enables the customization and refinement of the aliased contents too.

As an example, let's test it – modify the `dry/tomcat.yml` file as follows:

```
---
defaults:
  options: &tomcat_opts
    opts: '-Xms1G -Xmx2G'
    catalina_home: /opt/tomcat
instances:
  - name: instance-01
    options:
      <<: *tomcat_opts
      catalina_base: /opt/tomcat-01
  - name: instance-02
    options:
```

```
    <<: *tomcat_opts
    opts: '-Xms2G -Xmx2G'
    catalina_base: /opt/tomcat-02
```

so this time

- At line 3 we define the `&tomcat_opts` anchor with lines 4 and 5 as contents
- `*tomcat_opts` alias is merged to both the instances (line 9 and line 13)

Let's parse it with `yq`:

```
yq -y . dry/tomcat.yml
```

Validating Using Yamllint

As we have seen so far, despite being a quite straightforward human readable format, YAML has a lot of nuances that can lead to making mistakes when writing YAML documents.

To mitigate this and to make sure the YAML document is properly formatted, it is possible to use the `yamllint` command-line utility.

We can install it as follows:

```
sudo dnf -y install yamllint
```

To use it for validating a YAML document, simply run it followed by the path to the YAML file you want to check.

To have a go with `yamllint`, if it is not already installed on your system, we can install the `perl-CPAN-Meta` RPM package:

```
sudo dnf install -y perl-CPAN-Meta
```

This package contains, among its documentation, the `META-1_0.yml` file. This is an old, not properly formatted YAML file that we can use to run a `yamllint` scan returning warnings and errors.

Let's validate the `META-1_0.yml` file using `yamllint`:

```
yamllint /usr/share/doc/perl-CPAN-Meta/t/data-fail/META-1_0.yml
```

The scan generates the following output:

```
  2:2       warning  missing starting space in comment  (comments)
  3:1       warning  missing document start "---"  (document-start)
  3:14      error    too many spaces after colon  (colons)
  5:14      error    too many spaces after colon  (colons)
  8:35      error    too many spaces after colon  (colons)
  9:35      error    too many spaces after colon  (colons)
```

As you can see, `yamllint` is an invaluable companion tool which not only quickly identifies YAML syntax errors, but also helps you to give a consistent format to every YAML file. For this reason, the best practice is to always use it to validate the YAML document you write.

Processing YAML with Python

After having familiarized with YAML and the `yq` command-line utility that can be used as a brick to support YAML for shell scripts, we can now have a look at how to process YAML using Python.

A Poorly Written Example

The `tomcat` directory in the chapter's workspace contains the `tomcat.py` example Python script that loads its configuration from a YAML configuration file:

Let's have a look at the configuration file's contents:

```
cat tomcat/conf.yml
```

They are as follows:

```
---
!Config
defaults:
  !Defaults
  options: &tomcat_opts
    opts: '-Xms1G -Xmx2G'
    catalina_home: /opt/tomcat
```

```
instances:
  - !TomcatInstance
    options:
      <<: *tomcat_opts
      catalina_base: /opt/tomcat-01
  - !TomcatInstance
    options:
      <<: *tomcat_opts
      opts: '-Xms2G -Xmx2G'
      catalina_base: /opt/tomcat-02
```

As you can see, this YAML file contains the following custom tags:

- `!Config`
- `!Defaults`
- `!TomcatInstance`

As explained, these tags are used as identifiers to help the `tomcat.py` application in applying the logic necessary to process each different content.

Now let's have a look at the `tomcat.py` file:

```
cat tomcat/tomcat.py
```

A quick explanation of the lines of code is as follows:

- Lines 3 to 5 import the necessary modules.
- Lines 7 and 8 define the constructor function used to return the object generated from the specific YAML nodes.
- Lines 10 to 12 assign a constructor to each tag: this means that when the parser encounters the specified tag, it calls the specific constructor function to process its contents.
- Line 15 opens the `conf.yml` file for reading.
- Line 16 loads and processes the file contents using the PyYAML library.
- Lines 20 to 23 print the processed object in YAML format first and then in JSON format too.

Note Before running the script, make sure you have already installed PyYAML on your system; otherwise install it by running `sudo dnf install -y python3-pyyaml`.

Let's have a go with it:

```
cd tomcat
./tomcat.py
```

Maybe because of the eye-catching use of YAML tags and of the `try…catch` block, this script looks smart, but the sad truth is that this is a poorly written script: the aim was just to provide an easy-to-understand sample for starting.

The shortcoming with this script is that every tag calls the same constructor function that just loads the object in memory, without performing any kind of check on the structure or values.

Code like the above is very error-prone at runtime: it is a nightmare for those who will oversee maintaining the configuration, since it silently loads everything in the configuration file into memory. So, in the best-case scenario, it could just ignore mistyped settings, leaving the operator puzzled when not seeing their changes applied. In the worst-case scenario, it can crash the program as soon as it tries to use wrongly formatted data (and this may happen even long after program startup). So, from an operational perspective, this script can easily become unreliable, not to say a bomb.

Fancymonitor YAML support

As already done in the JSON chapter, we use the Fancymonitor project to demonstrate the proper way for processing YAML with Python.

Let's change to the Fancymonitor project's root directory as follows:

```
cd ~/devsecops-pillars/fancymonitor
```

Since YAML support has been added into the `f-yaml-config` branch, we must check out that branch:

```
git checkout f-yaml-config
```

The changes made to add the feature for supporting YAML configuration files can be easily shown by comparing the current branch (`f-yaml-config`) to the previous one (`f-json-config`).

The first interesting information is the list of modified files – we can get it as follows:

```
git diff --name-only f-json-config
```

Note Not so surprisingly, the list follows the same scheme we saw when checking how the support for JSON configuration files was added: having changes following a scheme means that the program's structure has been properly designed to promote easy improvements with minimal rework or even no rework at all. This is one of the benefits when spending some time drawing up a good, clean, and tidy design.

Now, as we did before, let's check the changes in each single file. For example, to see the changes in the `src/carcano/fancymonitor/core/configuration.py` file, run

```
git diff f-json-config -- \
src/carcano/fancymonitor/models/alert.py
```

Note Please repeat this check for each single file in the previous list.

As you can see, we modified every single object in the `models` directory to

- Import the `PyYaml` library
- Turn each single class into a class derived from `yaml.YAMLObject`:
- Add the `yaml_tag` attribute
- Define the constructor function specific for every class

These constructors are then called by the `yamlloader` function inside the new `yaml.py` file stored in the `loaders` directory.

We of course modified the `configuration.py` file in the core directory to try to parse and load the configuration file as a YAML document if parsing it as a JSON should fail.

Finally, the `main` function in the `__main__.py` file was modified to dump as a YAML the configuration object using the `yaml.dump` function.

In the `etc` directory, the `config.yml` file was also obviously added with the YAML configuration.

In the same way as before, to run it we must be in the `src/etc` directory and set the `PYTHONPATH` environment variable as follows:

```
cd src/etc
PYTHONPATH=.. python -m carcano.fancymonitor -c config.yml
```

To avoid loading the default configuration file, we use the `-c` command-line option to provide the path to the YAML configuration file.

This time it dumps the configuration in YAML format – if you fancy, you can even pipe it to `yq`.

Note Mind how now it can also be used to convert the `config.xml` file contents into YAML: just run `PYTHONPATH=.. python -m carcano.fancymonitor`. You can of course also turn the JSON configuration into YAML by running: `PYTHONPATH=.. python -m carcano.fancymonitor -c config.json`.

As you can see, being wise in designing and structuring things in a clean and organized way paid off: this improvement step was straightforward and didn't require any rework.

We just added new files and some additional code to the existing ones, but we didn't need to delete or rework anything.

Summary

In this chapter, we explored the YAML format in depth, starting with its semantics and advancing to topics like tags for effectively mapping YAML contents to objects and classes. We also covered YAML's DRY (Don't Repeat Yourself) features, including anchors, aliases, and the merge key. Additionally, we learned to validate YAML documents using `yamllint` to prevent syntactic errors and query them with yq, enabling YAML support in shell scripts.

By examining the YAML-related branch of the Fancymonitor project, we demonstrated how to process YAML documents in Python, leveraging tags with the `PyYAML` library for robust handling.

For DevOps and DevSecOps professionals, YAML expertise is crucial for configuration management and automation in modern software environments. Its human-readable structure makes it ideal for configuration files, deployment scripts, and data serialization. Understanding YAML facilitates defining and managing application settings, infrastructure as code, and CI/CD pipelines with clarity. Proficiency in `yq` further enhances this by offering command-line tools for querying, manipulating, and transforming YAML data, automating tasks, streamlining workflows, and ensuring configuration consistency. Together, these skills empower the creation of robust, maintainable, and scalable solutions, contributing to more effective software delivery and system reliability.

PART VI

Developing a Python Project

From a structural perspective, the Fancymonitor application we have worked with so far complies with software development best practices: its self-contained structure enables easy packaging (e.g., into a gzipped tarball), easing the deployment process.

Despite that, at the current stage, it still cannot be considered a Python project.

It is indeed still missing several facilities which a project must have, and operating it still requires too many manual steps. For example:

- Running it requires setting the `PYTHONPATH` variable to the directory containing the `carcano.fancymonitor` package.
- The packaging process is entirely manual.
- There is no installation script to perform an automated installation.

So, from a usability perspective, at this stage, the application is not suitable for the broad public – it is just good to be used by developers.

In addition to that, we are still missing three significant development best practices:

- Providing unit tests to granularly exercise its code unit by unit
- Using a build management system
- Using a packaging format specific to the development language and the target platform

The absence of these best practices and relying on many manual steps make everything error-prone. This has a substantial negative impact on the quality of the delivered software, since it prevents the software from being tested reliably. In addition to that, it also impacts the overall testing process, causing various issues that can lead to delays in any kind of delivery.

Moreover, it is still missing the following:

- Automated code style and code smell scans
- Security scans for vulnerabilities inside both the developed code and in the linked Python packages

Those are necessary to provide a consistent code that is easier to maintain across large teams as well as to deliver a safe product to the customer.

The chapters in this section show how to turn the source code we explored so far into a full-featured Python project, adding unit tests, implementing the necessary build management scripts, and packaging the application in Python's native formats and as RPM.

CHAPTER 14

Unit Tests

In this chapter, we delve into using PyTest to add unit test support to a Python project, generating Junit-formatted reports.

Skills on this topic are vital for DevOps and DevSecOps professionals, since it allows for automation in testing, facilitating continuous integration and deployment (CI/CD). Generating JUnit reports standardizes test results, making it easier to track code quality and security.

Overall, these practices contribute to a robust testing strategy that supports secure and reliable software delivery in a fast-paced environment.

What Are Unit Tests

Unit tests are a type of software testing that focuses on verifying the functionality of individual parts or "units" of code, typically functions or methods, in isolation from the rest of the application.

Implementing unit tests is a best practice fostered by many development disciplines and frameworks, such as Test-Driven Development (TDD), Behavior-Driven Development (BDD), Extreme Programming (XP), and the broadly known Agile methodology.

Why Use Unit Tests

The primary goal of unit tests is to ensure that each unit of code behaves as expected, producing the correct output for the supplied input.

These tests serve as an initial quality gate: if they don't pass, it indicates that the software has bugs, making it pointless to package the software and promote it to quality assurance. In such cases, the software cannot be advanced to the quality assurance stage.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_14

Testing the codebase with unit tests provides several benefits. For example, it mitigates and even prevents regressions: if a change breaks something, the unit tests will fail, alerting developers to the issue.

In addition to that, they also facilitate refactoring: with unit tests, developers have fewer concerns when refactoring, since they know they can run them to quickly check that the changes haven't introduced new bugs.

Unit tests bring also some side benefits. For example, since the code needs to be tested unit by unit, developers are encouraged to improve the modularity of the architecture of their software. They are also a good integration to the documentation, since they enable developers to verify the expected behavior by running them.

When to Use Unit Tests

To summarize, in general, it is often a winning choice to implement unit tests, especially for long-term projects or when dealing with critical and/or complex codebases that may be subject to frequent changes.

There are, of course, exceptions where you might skip developing unit tests, such as during prototyping or when writing short-lived code. Other exceptions include time constraints or low-risk areas, but these exceptions should be carefully evaluated, as they can lead to problems sooner or later.

Who Writes the Unit Tests

Because of the nature of unit tests, they are typically written by the developers who created the code being tested.

The PyTest Framework

When dealing with Python, PyTest is a third-party testing framework that has become popular due to its simplicity, powerful test discovery, and expressive syntax.

Compared to older frameworks such as `Nose`, it not only has a more intuitive and user-friendly syntax, but it also supports advanced features like parameterized testing and fixtures, which are either limited or not available in Nose.

Additionally, PyTest provides clearer and more informative failure messages, aiding in the quick diagnosis of issues. This makes it the preferred choice, especially when dealing with complex testing scenarios.

Install PyTest

The obvious prerequisite for running PyTest is installing it, so let's install it as follows:

```
sudo dnf install -y python3-pytest
```

Note The PyTest project is extensively documented, so explaining its syntax and usage here would not provide any real value: the information would quickly become obsolete and, for space constraints, not accurate enough. For this reason, I strongly encourage you to have a look at the PyTest official documentation available at https://docs.pytest.org/en/stable/contents.html.

A Real-Life Example

To help you to understand and to practice a little bit with PyTests, the `f-pytest` branch of the Fancymonitor project provides a few sample unit tests targeted at exercising the code lines used to process the XML, the JSON, and the YAML configurations.

To have a look at them, first change to the Fancymonitor project root directory:

```
cd ~/devsecops-pillars/fancymonitor
```

then, check out the f-pytest branch:

```
git checkout f-pytest
```

The changes in this branch from its parent are the files that have been added to provide PyTest unit tests to the Fancymonitor project.

The list of these files can be easily guessed by comparing the changes since the parent branch:

```
git diff --name-only f-yaml-config
```

The output is

```
src/test/pytest.ini
src/test/carcano/fancymonitor/bin/test_json.py
src/test/carcano/fancymonitor/bin/test_xml.py
src/test/carcano/fancymonitor/bin/test_yaml.py
src/test/carcano/fancymonitor/etc/invalid-values.yml
src/test/carcano/fancymonitor/etc/invalid-format.xml
src/test/carcano/fancymonitor/etc/invalid-values.yml
src/test/carcano/fancymonitor/etc/xsd-not-compliant.xml
src/test/carcano/fancymonitor/etc/xsd-not-found.xml
```

As you can see, files have been logically grouped to provide a clean and easy-to-follow structure.

Besides the `src/test/pytest.ini` file, which contains the option used when running PyTest, the rest of the files are grouped in two directories:

- `src/test/carcano/fancymonitor/bin` contains the Python files with the PyTest to run.
- `src/test/carcano/fancymonitor/etc` contains configuration files in different formats (YML and XML).

When started, PyTest automatically locates files having names starting with `test_` by walking the directory tree from the current working directory.

This branch provides three different test files with similar purposes but different targets:

- `test_json.py` runs JSON-related unit tests.
- `test_xml.py` runs XML-related unit tests.
- `test_yaml.py` runs YAML-related unit tests.

As an example, let's have a look at the contents of the `test_yaml.py` file:

```
cat src/test/carcano/fancymonitor/bin/test_yaml.py
```

As you can see, this unit test file, in addition to importing the `pytest` Python package, also imports the package containing the libraries to be tested.

Then, it defines three test functions:

- `config_yaml()`, which defines a fixture with a Fancymonitor config object loaded from a YAML file

- `test_config()`, which uses the `assert` function to check that the configuration in the above fixture has been loaded correctly
- `test_load_yaml_invalid_values()`, which loads the `invalid-values.yml` YAML file containing an invalid value to raise an exception on purpose. Since the exception causing PyTest to exit with failures is intentional, then the function has been decorated with the `pytest.mark.xfail` decorator to inform PyTest that the function is expected to fail on purpose. This causes PyTest to mark the test as `xfailed`.

The `test_json.py` and `test_xml.py` files work pretty much the same way: have a look at them too, to understand the full example.

In our implementation, it is mandatory to run the `pytest` command-line tool from the `src/test` directory: this requisite comes from having some of the PyTest files containing statements for reading files from relative paths. This requires running PyTest from a working directory that does not break these paths.

The last file to check is the `src/test/pytest.ini` file, which is used to provide the PyTest option to be used during the runs.

In this example, it just contains

```
[pytest]
addopts = --junitxml=../log/junit.xml
```

The `--junitxml` option is used to instruct PyTest to generate a Junit-formatted report in the provided path: in this case, it generates the `junit.xml` report file in the `../log` directory.

Note Despite not having its own RFC, JUnit XML format is widely recognized as a de facto standard for reporting test results, broadly accepted in the software development community, particularly in the context of JUnit 4 and its integration with tools. A JUnit XML report includes elements for test cases, suites, and results, providing detailed information about each test case, including success, failure, and error messages, which helps in diagnosing issues quickly. JUnit reports are often archived and compared over time, allowing teams to track test performance and stability across different builds identifying trends.

After exploring the provided files of this PyTest environment, it has now come the time to have a go at it. Change to the `src/test` directory, then simply run the `pytest` command-line utility:

```
cd src/test
PYTHONPATH=.. pytest
```

Note As you can see we are still required to manually specify the path to the Python libraries to load by setting the PYTHONPATH environment variable to point to the directory containing the `carcano.fancymonitor` Python package.

The output shows the progress while running the tests:

```
carcano/fancymonitor/bin/test_json.py.                  [ 10%]
carcano/fancymonitor/bin/test_xml.py xxx....            [ 80%]
carcano/fancymonitor/bin/test_yaml.py.x                 [100%]
```

Each. represent a passed test, whereas each x represents the xfailed ones. As anticipated, xfailed tests are failed tests that, since they are expected to fail, have been decorated with the pytest.mark.xfail decorator.

At the end of the run, it also generates a summary report:

```
=============== 6 passed, 4 xfailed in 0.41s ================
```

The summary reports six `passed` tests and four `xfailed`: since there are no `failed` tests, this PyTest run was successful.

If there were a real failure, an unsuccessful run would clearly mark the file with the failed test with a capital `F` character.

In such case, the output would be as follows:

```
carcano/fancymonitor/bin/test_xml.py xxF....            [ 80%]
```

The problem would also be reported in the summary, for example:

```
=========== 1 failed, 6 passed, 3 xfailed in 0.17s ===========
```

As you can see, the number of real failures is reported by the `failed` counter.

Note This is just a sample to show you the output to expect in the event of a real failure – it is not part of the lab.

Please note also how the output informs you that a JUnit report file has been generated, providing also the path to it.

As said, JUnit reports are XML files, so we can exploit our freshly acquired XML skill and use `xpath` to select only the `xfailed` tests, to get the rationale behind their outcome.

Since `xfailed` tests are marked in JUnit as skipped, just run

```
xpath -e "//testcase[skipped]" ../log/junit.xml
```

The output is as follows:

```
-- NODE --
<testcase classname="carcano.fancymonitor.bin.test_xml" name="test_
load_xml:xsd_not_found" time="0.002"><skipped type="pytest.xfail"
message="Expected failure when loading XML with a non-existent XSD path"
/></testcase>
-- NODE --
<testcase classname="carcano.fancymonitor.bin.test_xml" name="test_
load_xml:uncompliant_xsd" time="0.015"><skipped type="pytest.xfail"
message="Expected failure when loading XML not compliant to the XSD" /></
testcase>
-- NODE --
<testcase classname="carcano.fancymonitor.bin.test_xml" name="test_load__
invalid_format" time="0.002"><skipped type="pytest.xfail" message="Expected
failure caused by invalid formatted configuration file" /></testcase>
-- NODE --
```

As you can see, the JUnit reports are really a very convenient format for storing test reports.

As an exercise, investigate every expected-to-fail test to figure out why each of them is expected to fail: this is a very good exercise to have a closer look at how the example PyTests are implemented and how to exercise the exception handling logic of the code. A starting point is to remove the pytest.mark.xfail decorator and re-run the PyTest to see and understand the raised error.

Note A handy tool for processing JUnit reports is `junit-cli-report-viewer`, which provides a clear and concise summary of test results, highlighting both tests and their outcomes. Information about installation and usage are available in the project's homepage at https://github.com/amalfra/junit-cli-report-viewer.

Summary

In this short chapter, we discussed the significance of unit tests and how to implement them using the PyTest framework to ensure code reliability.

With an example setup generating JUnit reports, we not only learned how to read its output and summary report; we also discussed an example of clean and tidy structure for the tests' directory tree containing a few examples of unit tests, an approach which enhances maintainability and collaboration among team members.

Moreover, we learned how to turn failed tests into `xfailed` ones when they are expected to fail, providing also a rationale message that is then included in the JUnit report. This attitude helps teams to manage known issues without hindering deployment.

To summarize, incorporating unit tests and utilizing frameworks like PyTest are essential practices for DevOps and DevSecOps professionals. When integrated into a Continuous Integration pipeline, unit tests not only enhance code reliability but also facilitate early bug detection.

This fosters a culture of continuous testing and collaboration, which enables developers to create robust and secure applications that meet the demands of modern software development.

CHAPTER 15

Packaging As a Wheel

Software packaging is a crucial step in the DevOps lifecycle, not only in delivering applications to end users but also in facilitating effective collaboration with testing teams. Proper packaging ensures that the software is structured and ready for deployment, which is essential for maintaining quality and security standards.

When dealing with packaging software sources, it is important to also include unit tests in the package, so that the CI/CD using it to automate the package creation for end users can run them during the packaging process, aborting the process in case of failures.

This integration of testing is vital for mitigating the risk of issues reaching the end users, thereby enhancing the overall reliability and security of the software. Collectively, this contributes to streamlining workflows and improving quality assurance, resulting in a more secure software delivery process.

The Importance of Packaging

While the self-contained structure of the Fancymonitor application makes it suitable to be packaged into a gzipped tarball, this approach represents a suboptimal distribution method that introduces several significant limitations compared to proper packaging systems.

Gzipped tarballs indeed lack some basic features you would expect from a distribution package format, for example:

- The contents of the archive are not automatically extracted to the proper path used by Python for storing its packages, such as the system-wide or user-specific Python `site-packages` directory.
- During the extraction, they do not generate a Python driver script that allows users to easily run the application.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_15

These shortcomings necessitate additional manual steps for users, including the installation of the application packages into the correct directories and the creation of a wrapper shell script. This wrapper script serves to simplify the process of launching the application by encapsulating the command needed to run Python and load the application module: for example, instead of typing the full command (e.g., `python3 -m carcano.fancymonitor`), supposing the wrapper script is called `fancymonitor`, users can simply type `fancymonitor` to start the application.

Furthermore, to install an application system-wide, it is necessary to place files in standard locations within the Linux filesystem, such as placing configurations under the `/etc` directory tree and system-wide executable scripts under `/usr/bin`.

While all the above tasks can be done manually, this process is cumbersome, slow, error-prone, and quickly turns into a maintenance nightmare.

These shortcomings also affect installations in testing environments: delivering an application to the testing team with an error-prone and flaky installation process risks leading to unreliable test outcomes generated by mismatches and leftovers from the previous installation. Proper packaging instead promotes consistency and reliability by delivering the software in a consistent state, including automatic resolution and installation of all the dependencies, proper configurations, and environment settings.

This ensures that the packaged application can be reliably reproduced, which is crucial for debugging and for ensuring that the version of the tested software is consistent, along with all its dependencies.

That's why packaging the software is always best practice, even when the software is intended for internal testing.

A peculiar packaging type is containers: they enable running applications in isolation from other software on the system, reducing the risk of conflicts and ensuring that the host environment remains clean and controlled; this makes them particularly suitable also in the testing field, where different applications of different versions (and so potentially with different requirements) get continuously deployed and tested.

Note If necessary, packaging provides an easy way to vendor the required libraries, ensuring that the application always runs with the expected library versions. Be wary that while this practice mitigates the risk of missing or incompatible versions of external components, it has the cumbersome tradeoff of requiring you to release a new software version every time a vulnerability is found in any of the vendored libraries. This is, for example, the pain point of deploying applications using containers.

Python Packaging

Python provides packaging guidelines through the Python Packaging Authority's (PyPa): this is a community-driven organization that plays a crucial role in the development and maintenance of packaging standards and tools in the Python ecosystem.

PyPa is responsible for defining best practices, guidelines, and specifications that help developers create and distribute Python packages effectively.

Setuptools

Setuptools is part of the Python Packaging Authority's (PyPa) efforts to standardize and improve the packaging ecosystem.

It is a Python package development and distribution library that provides tools for packaging Python projects, including the creation of source distributions (`sdist`), binary distributions (`bdist`), and managing dependencies.

It can be configured using two files: `setup.py` and `setup.cfg`. As the name suggests, `setup.py` is a Python-formatted file, while `setup.cfg` is an optional declarative configuration file.

These setup files are used to specify information about the package, such as which files to include, the minimum required Python version, dependencies, supported testing suites, and various metadata used to classify the package within the Python Package Index (PyPI).

The command-line syntax for invoking the creation of a Python package is

```
python setup.py <command>
```

where `<command>` can be `sdist`, `bdist`, or `bdist_wheel`.

Note While `setup.py` remains a valid configuration file, the direct invocation of `setup.py` is now deprecated: the community encourages using the new Python `build` module instead.

The Python Build Module

The introduction of the Python `build` module is part of ongoing efforts to adhere to the standards set by `PEP 517` and `PEP 518` recommendations, which aim to improve the packaging ecosystem.

The `build` module, like `setuptools`, is part of the Python Packaging Authority's (PyPa) efforts to standardize and improve packaging.

It provides a simple command-line interface to build Python packages from source, enabling the creation of both source distributions (`sdist`) and built distributions (`wheel`) based on the configuration specified in the `pyproject.toml` file.

This file, introduced by `PEP 518`, specifies build system requirements for Python projects and allows projects to declare their build dependencies and configuration in a standardized manner.

Among the various settings, `pyproject.toml` instructs the `build` module about which build backends to use.

The available backends include `setuptools`, `poetry`, `flit`, `hatch`, and `pbr`.

When dealing with new projects, the most commonly used tools are `flit` for simpler packages and `poetry` for more complex ones.

Setuptools remains the most broadly used for existing projects, primarily because many projects have already leveraged it and it has a long history of support and documentation.

Note When starting new projects, you can easily install poetry by running as `root` user `curl -sSL https://install.python-poetry.org | POETRY_HOME=/opt/poetry python3 -`. Once installed, you can use it to generate and manage new projects.

The Python `build` module is available as an RPM package – since we are about to see it in action, let's install it as follows:

```
sudo dnf install -y python3-build
```

Configure the Build Module in an Existing Python Project

To help you learn how to exploit `setuptools` and the `build` module in your own Python project, the Fancymonitor project provides a good example you can use as a reference.

The relevant files have been added into the `f-setuptools` branch.

To see it in action, change to the Fancymonitor project's directory as follows:

```
cd ~/devsecops-pillars/fancymonitor
```

then, check out the `f-setuptools` branch:

```
git checkout f-setuptools
```

The `pyproject.toml` file is contained in the `src` directory.

Open it and invest some of your time carefully examining its contents:

```
cat src/pyproject.toml
```

The content of this file is straightforward and self-explanatory, so I won't dig into details that can be easily guessed.

The only point I want to remark is the following snippet:

```
[build-system]
requires = [ "setuptools", "wheel" ]
build-backend = "setuptools.build_meta"
```

Besides listing the Python packages required for the build, it sets `setuptools` as the build backend: this should be expected, since we are dealing with already existing Python code.

Note The best source for learning the pyproject formats and in general how to package a Python project is the Python Packaging User Guide. The section related to the `pyproject.toml` file is currently available at https://packaging.python.org/en/latest/guides/writing-pyproject-toml.

Including Tests in the Distribution

By default, PyTest unit tests are not included in the distribution package. This is normally correct, since they are not really necessary for the end user.

In this book, we will also see later how to pack the generated Python distribution package into an RPM package. Since the RPM build process has also a section for automatically running unit tests, aborting the build in case of failures, we will force the inclusion of the PyTest unit tests in the source Python distribution package.

To achieve this, we exploit the `src/MANIFEST.in` file: this file is used to specify which files to exclude and which ones to include in the distribution of a package.

Let's check its contents:

```
recursive-include test *
exclude test/**/__pycache__/*
```

So, in our example, it states to include the `test` directory tree, excluding the `__pycache__` subdirectories (if any), since they contain compiled Python files that are not necessary for distribution.

By using this trick, every file related to PyTest, including configuration files (XML, JSON, and YAML), is added to the distribution package.

Note A word of caution: in my experience it is best to include unit test files to the source distribution only. Although there is a way for doing this directly relying on `setup.py` and/or `pyproject.toml`, my practical experience was always disappointing: depending on the Python and on the used libraries' version, sometimes it adds them unpredictably to both the dist and wheel or doesn't add them at all. For this reason, I am suggesting the use of the `src/MANIFEST.in`. The drawback of this trick is that you must remove the tests on the fly before building the wheel distribution package, since it is not supposed to contain unit tests – we'll see how to easily automate this in the next chapter when talking about GNU Make.

The changes in this branch do not involve only providing the `pyproject.toml` and `src/MANIFEST.in` files:

```
git diff --name-only f-pytest
```

As you can see, there are also a few additional resources:

- The application's documentation (`src/doc/README.md`): In this case, it is just a stub; a proper readme file must follow the guidelines described at https://google.github.io/styleguide/docguide/READMEs.html.
- The Python's standard logging facility configuration file (`src/etc/logging.conf`): Currently set to log to the `fancymonitor.log` file beneath the `log` directory.
- The empty log file (`src/log/fancymonitor.log`).

Note Every single object and function of the Fancymonitor application contains logging statements using the Python logging facility. You can alter the severity by setting the `LOGLEVEL` environment variable. If omitted, it logs with a severity equal to `INFO`. Valid values are shown in the definition of the `logging_levels` variable in the `__main__.py` file.

Providing all the necessary settings files and documentation bundled with the package is considered best practice: it does not matter if the target users are software testers or the final customers, they will certainly benefit from this.

Mind that it is the developers' responsibility to write and provide settings and documentation files.

Build a Source Distribution

After understanding how the Fancymonitor project has been configured for packaging, we can now see the process for generating the deployment packages.

First, change to the `src` directory:

```
cd src
```

then, to generate a source distribution, run

```
python3 -m build --sdist
```

At process completion, the `dist` directory contains a `.tar.gz` file: this is the gzipped tarball archive with the package's sources.

You can check its contents by running `tar tfvz` followed by the archive file path. As you can see, it contains also the PyTests files.

Build a Wheel Distribution

The source package we built is not intended to be distributed to the end users. The best Python distribution package for the end user is the `wheel` distribution.

We can generate this project's Python `wheel` package by running:

```
python3 -m build --wheel
```

At process completion, the `dist` directory contains a `.whl` file. This is Python's `wheel` package: since it is a zipped archive, if you fancy you can inspect its contents by running `unzip -l` followed by the path to the `wheel` file.

If you inspect the wheel's contents, you'll notice it contains also the unit tests: as said, this is a side and undesired but expected effect, caused by the trick we did with the `src/MANIFEST.in` file.

Python wheels are application distribution packages targeted to the end user, so they are not supposed to ship unit test files.

For this reason, we must use another trick to remove the test directory tree before building the `wheel`, restoring it immediately after that.

To see the trick in action, we must first wipe out the build directory to avoid any leftovers of the builds we have already run so far to be accidentally included in the `wheel` package.

```
[ -d build ] && rm -rf build
[ -f dist/*.whl ] && rm -f dist/*.whl
```

The trick consists of running in sequence the following three statements:

```
tar --exclude='test/.pytest_cache' -zcf test.tgz test \
&& rm -rf test
python3 -m build --wheel
tar xfz test.tgz && rm test.tgz
```

The first statement, after saving it to the `test.tgz` archive, removes the `test` directory, preventing it from being included in the generated Python package. After the second statement generates the Python wheel package, the last statement restores the test directory and removes the `test.tgz` archive.

This multistep process must always be run each time we need to build the wheel package.

Multistep repetitive tasks are cumbersome and error-prone: the only way to mitigate these drawbacks is of course automating them. We'll see how to deal with this in the next chapter, when talking about automating build management by using GNU Make.

Summary

In this chapter, we explored the critical role of packaging developed software before delivery. This step is essential not only for final distribution to end users but also for ensuring quality during testing phases.

We covered practical techniques for packaging Python projects using `setuptools` and the `build` module, applying them in a real-world scenario with the Fancymonitor application. Additionally, we learned how to integrate unit tests into Python source packages, for example, to be able to perform automated testing while packaging in other formats, such as an RPM package.

Packaging proves invaluable for all stakeholders: it saves time, minimizes installation errors, and facilitates seamless version management, including upgrades and downgrades. For DevOps and DevSecOps professionals, mastering Python project structures and packaging is key to efficient dependency management, streamlined deployments, and maintaining secure, reliable, and maintainable applications throughout their lifecycle.

CHAPTER 16

Managing the Build with GNU Make

In this chapter, we explore the significance of using build management tools to govern the build process, a critical skill for DevOps and DevSecOps professionals. These tools automate repetitive and error-prone tasks, boosting efficiency and speed in the build and test pipeline. By doing so, they accelerate the development cycle while ensuring testing integrity, preventing issues from improper installations or leftover artifacts from prior builds.

To demonstrate how these practices enhance software quality, we examine a practical example: integrating GNU Make into the Fancymonitor project. The project's `Makefile` defines targets for building both source and complete Python packages, with dependencies that trigger PyTest runs.

Upon completing this chapter, readers will gain the essential concepts and skills to create fast, reliable, and effective pipelines. This empowers them to deliver more robust software, fostering a culture of continuous improvement and security throughout the development lifecycle.

The Importance of a Build Automation Tool

At this stage of the development, the Fancymonitor is no longer just a bunch of Python files: it has configuration files, resource files, and documentation, and now it is also packageable using the Python `build` module, but as we saw this process still requires some manual steps, making it cumbersome and error-prone.

This is normal and should be expected; it is just the complexity of the project gradually growing.

We have reached enough complexity to justify the effort to add a build automation tool: this enables us to easily trigger these processes by running single statements.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_16

Using a build management tool brings several benefits, such as

- **Improved consistency**: The build, test, and run process is consistent across the whole toolchain.
- **Human error prevention**: Its automated tasks spare developers and QA engineers from manually running error-prone repetitive statements. This results in a faster and more effective delivery.
- **Scalability**: As projects grow, managing builds manually becomes cumbersome. A build management tool helps keep this under control.
- **Documentation**: A build management tool is anyway somehow an indirect documentation, since anybody interested in specific topics can just check how they are addressed by reading the statements.

What Is GNU Make

GNU Make is an old-fashioned yet still valuable build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles.

Conversely from other specialized build automation tools such as ANT or Maven, GNU Make is a general-purpose one that provides a lot of flexibility with a far smaller complexity and learning curve. Of course, being generic has also drawbacks, such as, since it is not language specific, it cannot have features such as automatic dependency resolution and download of required packages.

Despite being originally designed to manage the build process of compiled applications, it is flexible enough to also fit scripted application use cases and to automate the management of tasks such as running unit tests, packaging the application, digitally signing the application, pushing the application to a distribution repository, and so on.

This makes this tool an invaluable brick of CI/CD pipelines.

Installing GNU Make

Unless you selected the `Development Tools` package group at installation time, then GNU Make is not already installed on your system.

Since we are about to run a small lab with it, please install it as follows:

```
sudo dnf install -y make
```

Makefiles

As said, GNU Make relies on `Makefiles`: these are special files that define, grouped by `target`, a set of tasks to be executed.

The basic structure of a `Makefile` includes

- **Targets**: They are a set of tasks to be run to accomplish a specific purpose. They are so flexible to be suitable for plenty of use cases, … just to make a few examples: building an application or a specific part of it, assembling several parts into a runnable application, running unit tests, running static code analysis scans, generating documentation, packaging a built application, and publishing the package into a repository such as a PyPI compatible repository, a DNF/YUM repository, and so on.
- **Dependencies**: Each target lists the other targets it depends on, to have them run automatically when it is invoked.
- **Statements**: The set of statements to run in each target – these are just shell statements.

Inside a `Makefile` it is possible to define custom rules and variables, allowing for flexible build configurations: this means that within the `Makefile` you can easily integrate other tools and scripts, allowing for complex build processes that may involve testing, packaging, and deployment.

Note I suggest you have a look at the official GNU Make documentation available at https://www.gnu.org/software/make/manual/make.html.

A Real-Life Example

To provide you with a practical lab with GNU Make, a `Makefile` and a few additional resources have been added to the Fancymonitor's `f-make` branch.

Note In the Fancymonitor-specific scenario, since Python is a scripted language, we don't need to go through the process of building from sources. In any case, keep in mind that when dealing with compiled software, a build management tool is a valuable ally, as it allows for quick partial recompilation to rebuild only the modified part of the software, automatic compilation of dependencies, and many other features that save a lot of time and prevent human errors.

Change to the project's Python directory as follows:

```
cd ~/devsecops-pillars/fancymonitor
```

then, check out the `f-make` branch:

```
git checkout f-make
```

Let's have a look at the `Makefile` so that we can discuss it:

```
cat src/Makefile
```

This `Makefile` contains the following targets:

- `clean`: It removes any leftovers from previous runs. Targets like this, which do not generate actual contents, are called phony targets.
- `release`: A phony target that validates the format of the `RELEASE` shell environment variable, failing if it is invalid. If the format is good, it sets it in the `version` field of the `pyproject.toml` file.
- `dist`: After running the `clean` and the `release` targets, it generates the source distribution package.
- `wheel`: After running the `test` and the `release` targets, it generates the Python wheel package.
- `test`: After running the `clean` target, it runs every defined PyTest.

- `install`: Installs Fancymonitor in the system – please remember that its usage is twofold.
  - **User-level install**: The basic usage installs `carcano.fancymonitor` as a user package and creates the `~/fancymonitor` directory with a self-contained structure, containing the `bin`, `etc`, `doc`, and `log` directories. Once installed, it is possible to run the application from the `~/fancymonitor/bin` directory by launching the `fancymonitor` shell script. Since it does not require administrative privileges, this kind of installation is particularly useful for developers and test engineers who need to autonomously install and reinstall the application.
  - **Chrooted install**: As its name suggests, a chrooted installation generates a directory tree by changing the root directory for the installation process. This method is typically used when creating RPM packages, as it allows for the recreation of a Linux filesystem structure within the chroot directory. This isolated environment serves as a reference structure during the packaging of files. To perform a chrooted installation, you need to set the `ROOT` variable to the desired chrooted path. It is common practice to leave the `PREFIX` variable empty, as the installation occurs within the `ROOT` directory rather than a standard system path. Additionally, the variables `DOCDIR`, `CONFDIR`, `SHAREDSTATEDIR`, and `LOGSDIR` should be set as absolute paths, specifying where documentation, configuration files, shared state files, and logs will be placed within the chrooted environment.

As a learning exercise, after changing to the `src` directory, try running every target one by one:

- `make clean`
- `make release`
- `make dist`
- `make wheel`
- `make install`

Note Since it is required by this `Makefile`, remember to export the `RELEASE` variable with the release version you want to generate before running the `release` target or any target even indirectly dependent on it. For example: `export RELEASE=0.1.0`.

Since we have run `make install` without setting any variable, it performed a user-level install resulting in having Fancymonitor installed in your home directory.

Let's try running it:

```
cd ~/fancymonitor/bin
./fancymonitor
```

To go on with this GNU Make small lab, change back to the project's `src` directory as follows:

```
cd ~/devsecops-pillars/fancymonitor/src
```

The dependencies specified for each single target enable you to reduce the number of statements to reach a specific target.

For example, to generate a wheel Python package, just run:

```
export RELEASE=0.1.0
make wheel
```

The `clean`, `release`, and `test` targets are automatically invoked, and if any unit test fails, the whole process is aborted.

Summary

In this chapter, we explored the importance of using build management tools to govern the build process, enhancing efficiency by automating time-consuming and error-prone tasks. This automation accelerates the pipeline while safeguarding testing integrity, preventing flawed outcomes from improper installations or remnants of previous builds.

We examined a real-world example using GNU Make to create targets for building both source and complete Python packages, with dependencies that run PyTests. This setup boosts software quality by halting the packaging process if unit tests fail.

The concepts covered equip DevOps and DevSecOps professionals with tools to automate and streamline build management, foster collaboration, integrate security practices, and ensure consistent builds. Ultimately, these skills enable the timely delivery of high-quality, secure software.

CHAPTER 17

Packaging As RPM

This chapter explores additional essential concepts for DevOps and DevSecOps professionals in application packaging, with a focus on the advantages of using system-specific packages like RPM (for Red Hat-based systems) or APT (for Debian-based systems). These formats enable seamless integration with the operating system's package manager, enhancing overall system maintainability and security.

Familiarity with these packaging formats equips professionals to build reliable, consistent packages. For example, understanding RPM packaging – including the structure and use of SPEC files – is crucial for creating deployable artifacts that integrate smoothly with system tools. To demonstrate this specific use case, this chapter provides a step-by-step guide to packaging the Fancymonitor Python project as an RPM.

This chapter also provides some practical examples of how to operate with the RPM package manager: it is indeed equally important learning how to query package metadata, contents, and dependencies. This knowledge empowers practitioners to manage software deployments more effectively, ensuring applications are installed securely, tracked accurately, and maintained consistently across diverse environments.

Note Before proceeding with the chapter, we must set up the necessary environment by creating a dedicated `build` group and assigning it as a secondary group to your current system user. This group will be required for several labs later in the book, so do not skip this step. Just run `sudo groupadd build; sudo usermod -aG build $(whoami); exit`. Once done, log back in or restart your session to apply the group changes.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_17

Platform-Specific Packaging Formats

Wheel is a packaging format specifically crafted for the Python programming language, designed to meet Python's distribution requirements. When dealing with broad deployments, it is generally best to use platform-specific packaging formats whenever possible, such as RPM for Red Hat family distributions or DEB for Debian-based distributions.

While this may require additional packaging effort, it provides significant benefits in terms of system maintainability, as these formats are fully integrated into the package management systems of their respective Linux distributions, such as DNF/YUM for Red Hat and its derivatives. This deep integration results in faster deployments and reduced error rates during patching operations, ultimately lowering maintenance costs and increasing service availability.

Moreover, platform-specific packaging also offers substantial security benefits: besides enabling to easily locate files provided by each single package and vice versa, their package managers typically provide easy access to additional information and features that can be leveraged to quickly identify Common Vulnerabilities and Exposures (CVEs) that the installed packages may be affected by.

Red Hat Package Manager

RPM was developed by Red Hat in the mid-1990s to simplify the management of software packages on Linux systems. Its primary goal is to facilitate the installation, updating, and removal of software packages.

While RPM is most associated with Red Hat-based distributions (like Red Hat Enterprise Linux, Fedora, and CentOS), it is also used by other distributions, including SUSE and Oracle Linux. Even at CERN they had their own RPM-based Linux distribution, called Scientific Linux. This has led to a broader adoption of the RPM format and tools.

RPM maintains a database that contains metadata about installed packages. This metadata includes not only basic information (like package name, version, checksum, and vendor) but also more detailed information, such as file lists, dependencies, and library versions. This allows the package manager to handle complex dependency resolution and ensure that all required components are present and at the right version.

The information stored in this database is brought by the packages themselves: to accomplish this Red Hat developed also the RPM package format.

So, when people say "RPM", the actual meaning depends on the context: that word may indeed refer to either the package management system or the packaging format.

RPM is operated by using the `rpm` command-line tool … we will see a few usage examples later in this chapter.

The RPM Packaging Format

The RPM format is a sophisticated archive that not only packs a set of files and directories within a package file but can also run scripts and evaluate conditionals during installation or removal. It is structured by placing a header on top of a CPIO archive.

The package itself has four main sections:

- **Magic number**: A file identifier used to recognize the file contents as an RPM package
- **Signature**: A section that verifies the integrity and authenticity of the package
- **Header**: Tagged data containing package information, such as name, version numbers, dependencies, and copyright
- **Payload**: The actual CPIO archive containing the program files and directories

Since RPM packages are based on the CPIO format, it is even possible to list their files without using the `rpm` command-line tool.

As an example, let's download the `aide` RPM package to the `/tmp` directory as follows:

```
sudo dnf install -y --downloadonly --downloaddir=/tmp aide
```

We can list the contents of the `epel-release-8-13.el8.noarch.rpm` package by running

```
rpm2cpio /tmp/aide-*.rpm | cpio -i --list
```

If necessary, we can even extract the files into the current directory by running

```
mkdir /tmp/aide; cd /tmp/aide
rpm2cpio /tmp/aide-*.rpm | cpio -idv
```

These commands allow us to inspect or extract files from RPM packages without even needing to install them.

Building RPM Packages

RPM packages are generated by the `rpmbuild` command-line utility by parsing a manifest containing the necessary directives called `SPEC file`.

The `rpmbuild` command is provided by the `rpm-build` RPM package. Since it is not installed by default, we must install it as follows:

```
sudo dnf install -y rpm-build
```

To illustrate the RPM build process with a practical example, we will use the SPEC file provided by the `f-rpm` branch of the Fancymonitor project.

This SPEC file has been tailored to generate two RPM packages from the Fancymonitor's Python wheel package.

Since building RPM containing Python code requires specific RPM macros, it is also needed to install the `python3-devel` RPM package providing them.

```
sudo dnf install -y python3-devel
```

After installing all the necessary tools to build RPM packages, we can now change to the Fancymonitor project directory:

```
cd ~/devsecops-pillars/fancymonitor
```

and check out the `f-rpm` branch:

```
git checkout f-rpm
```

As we already did multiple times, let's list the changes since the parent branch (`f-make` in this case):

```
git diff --name-only f-make
```

The output is

```
Makefile
src/Makefile
RPM/SPECS/carcano_fancymonitor.spec
```

As you can see, besides the `src/Makefile` we discussed in the previous chapter, this branch contains also another `Makefile` in the project's root directory. This is used to provide the `rpm` target to launch the RPM build process.

Let's have a closer look at it:

```
cat Makefile
```

As you can see, it defines four targets: `clean`, `dist`, `rpm`, and `all`, which is the default target.

The first two just contain the following statement:

```
$(MAKE) -C src $(MAKECMDGOALS)
```

This command runs the `make` command in the directory specified by the `-C` option, using the same target as the current invocation. In other words, it invokes the specified target in the existing `src/Makefile`.

The interesting one is the `rpm` target – it implements the necessary logic to build the RPM running a few steps in the following order:

- Build the Python's dist package because of the `dist` target dependency.
- Run its own logic generating the `rpmbuild` directory tree in the user's home directory and moving the Python's dist package generated while running the `dist` target into the `~rpmbuild/SOURCES` subdirectory. This is the path where the `rpmbuild` command-line tool expects to find the source archives to process.
- Copy the SPEC file to the `~rpmbuild/SPECS` subdirectory.
- Run the `rpmbuild` statement using the `RPM/SPECS/carcano_fancymonitor.spec` SPEC file.

Note Since the new `Makefile` contains the `rpm` target, we needed to add a target with the same name also to the already existent `src/Makefile`.

The Spec File

Let's then have a quick overview of the contents of the `RPM/SPECS/carcano_fancymonitor.spec` SPEC file:

```
cat RPM/SPECS/carcano_fancymonitor.spec
```

As you can see, the first part contains the package's metadata; then it starts grouping statements by section – the beginning of a section is marked by a line starting with an identifier prepended by the % character.

There are also several macros, such as `%autosetup`, which is used to automatically guess a way to extract the contents from the sources archive.

Table 14-1 summarizes the sections in this SPEC file.

Table 14-1. *RPM SPEC file sections*

Marker	Section
%description	This section provides a description of the RPM package. Each subpackage can have its own description section, identified by the `%description` marker followed by `-n` and the subpackage's name.
%package	This section declares a subpackage and allows you to provide specific metadata for that subpackage. It is typically followed by a `%description` section for the subpackage.
%prep	This marks the beginning of the **preparation phase**, where the build environment is set up, often by unpacking the source archive into the build directory. The path to the build directory can be inferred from the `%{_builddir}/%{srcname}-%{_version}` macros. A best practice here is to enforce idempotency by purging the build directory each time it is already set.
%build	This marks the beginning of the **build phase**, where the software is compiled. The best practice is to include a `make` statement with the correct target. In this case, we also use a `sed` command before removing the test target dependency from the wheel target, as we want to trigger the test target alone during the `%check` phase.

(*continued*)

Table 14-1. (*continued*)

Marker	Section
%check	This marks the beginning of the **check phase**, where the software is validated by running unit tests. A failure in this phase prevents the RPM build from completing. Again, the best practice is to include a make statement with the appropriate target.
%install	This section marks the beginning of the **installation phase**, where commands to install the built software into the build root directory are specified. Also, here the best practice is to include a `make` statement with the correct target, which is run by the `%make_install` macro. Note how we export the variables to perform a chrooted installation before running it. Please, mind also how the main application files (such as the main binary, configuration files, and documentation) have already been copied to the right directories during the installation process by the Makefile `install` target. Since the copy process leaves them there too, they become leftovers: to address this, in this section we also added the necessary `rm` statements to remove them from the chrooted Python's `sitelib`. Leaving them there, since they are not included in the RPM, would result in an "unpackaged files" error.
%files	This section lists the files that must be bundled in the package, along with their POSIX permissions and the owning user and group to set during installation. Each subpackage can also have its own `%files` section, identified by the `%files` marker followed by `-n` and the subpackage's name.
%changelog	This section marks the beginning of the changelog, which tracks important changes and updates for each release of the package.

There are also other sections, for example, where to place scripts to run before or after installing or uninstalling, but they are not relevant for the Fancymonitor application.

Note Before going on with the reading, inspect the SPEC file and make sense of how it works. You are also encouraged to read the official RPM SPEC file documentation at https://rpm-software-management.github.io/rpm/manual/spec.html.

Building the RPM

We can finally have a go and build the Fancymonitor RPM package.

First things first, since it is required by the `dist` target, we must export the `RELEASE` environment variable to specify the release version we want to build:

```
export RELEASE=0.1.0
```

Then, simply run `make` with the `rpm` target:

```
make rpm
```

As indicated by the build output, the `dist` target is triggered first as a prerequisite, generating the `carcano_fancymonitor-0.1.0.tar.gz` Python source distribution package in the `src/dist` directory.

The process then continues by installing the `~/rpmbuild/SOURCES/carcano_fancymonitor-0.1.0.tar.gz` into the `~/rpmbuild/BUILDROOT` directory.

Depending on your platform, the installation subdirectory is either

- `python3-carcano-fancymonitor-bundle-0.1.0-1.el9.x86_64`
- `python3-carcano-fancymonitor-bundle-0.1.0-1.el9.aarch64`

The next step, since the SPEC file contains the `%check` section, is running the PyTest tests. Please mind that, to have it working, we intentionally added them to the Python source distribution packages with the purpose of running them also while building the RPM package.

The output indeed contains

```
carcano/fancymonitor/bin/test_json.py .                  [ 10%]
carcano/fancymonitor/bin/test_xml.py xxx....             [ 80%]
carcano/fancymonitor/bin/test_yaml.py .x                 [100%]
```

Note Any failure at this stage will abort the RPM build process.

In the final step, the contents installed in the chrooted directory are assembled into the final RPM packages.

At the end of the build, the `~/rpmbuild/RPMS/noarch` directory contains the following `noarch` RPM packages:

- `carcano-fancymonitor-0.1.0-1.el9.noarch.rpm`
- `python3-carcano-fancymonitor-0.1.0-1.el9.noarch.rpm`

As you can see, filenames are clearly derived from the subpackages defined in the SPEC file.

Meanwhile, the `~/rpmbuild/SRPMS` directory contains the source RPM (SRPM) file, which can be used to distribute the source Python package and the SPEC file, allowing you to replay the build process on other systems as needed.

To see this last concept in action, let's first remove the directory trees containing the `carcano_fancymonitor-0.1.0.tar.gz` and the `carcano_fancymonitor.spec` files.

```
rm -rf ~/rpmbuild/{SOURCES,SPECS}
```

Let's now get them back by installing the source RPM package:

```
cd ~/rpmbuild/SRPMS
rpm -ivh \
python3-carcano-fancymonitor-bundle-0.1.0-1.el9.src.rpm
```

If we now check the contents of the directories we just deleted

```
ls ~/rpmbuild/{SOURCES,SPECS}
```

we find that both the directories have been re-created and that both `carcano_fancymonitor.spec` and `carcano_fancymonitor-0.1.0.tar.gz` files have been restored.

Note The RPM packages generated are not GPG signed: to avoid tampering, the best practice requires using GPG to sign the RPM packages – we will discuss this in detail in Chapter 22, which is entirely dedicated to GPG.

Operating with RPM

Before installing the generated RPM packages, we use them to illustrate a few example statements with the `rpm` command-line tool to get information on the RPM packages.

First, navigate to the RPMS directory containing them:

```
cd ~/rpmbuild/RPMS/noarch
```

The very first and most common use case is querying the RPM package to retrieve metadata information. The options to use are

- The `-q` option, to enable the query feature
- The `-i` option, to ask it to print the metadata information

For example:

```
rpm -qi carcano-fancymonitor-0.1.0-1.el9.noarch.rpm
```

Note As an exercise, repeat the same command also for the other `noarch` and for the source RPM file.

Another very common need is listing the files provided by the RPM packages. In this case, in addition to the `-q` option, we must use the `-l` option to ask for a listing of the RPM contents:

```
rpm -ql carcano-fancymonitor-0.1.0-1.el9.noarch.rpm
```

The output reveals that this package provides

- Configuration files in `/etc/fancymonitor`
- The main application script at `/usr/bin/fancymonitor`

- Documentation in `/usr/share/doc/fancymonitor-0.1.0`
- An empty log file in `/var/log/fancymonitor`

We can try installing it as follows:

```
sudo rpm -ivh \
carcano-fancymonitor-0.1.0-1.el9.noarch.rpm
```

The operation fails complaining that it is missing a dependency:

```
error: Failed dependencies:
    python3-carcano-fancymonitor is needed by carcano-fancymonitor-0.1.0-1.
el9.noarch
```

We should expect this, since in addition to the `carcano-fancymonitor` RPM package, the build process also generated the `python3-carcano-fancymonitor` RPM subpackage. In addition to that, when we listed the contents of the `carcano-fancymonitor-0.1.0-1.el9.noarch.rpm` RPM, it was missing every Python file.

For the sake of completeness, mind that it is possible to list the requirements of an RPM package without attempting to install it. The option to provide in this case is `-R`. For example:

```
rpm -qR carcano-fancymonitor-0.1.0-1.el9.noarch.rpm
```

The output confirms that this package requires `python3-carcano-fancymonitor`.

In our case, we already know that the `python3-carcano-fancymonitor` component is provided by the other `noarch` RPM package. In real life instead, to address dependency solving problems, you often need to query RPM packages to check what they provide.

This can be accomplished by querying the RPM package using the `-P` option as follows:

```
rpm -qP python3-carcano-fancymonitor-0.1.0-1.el9.noarch.rpm
```

The presence of the `python3-carcano-fancymonitor` among the output lines confirms that the `python3-carcano-fancymonitor` component is provided by this RPM package.

For the sake of completeness, let's also have a look at the files it provides:

```
rpm -ql python3-carcano-fancymonitor-0.1.0-1.el9.noarch.rpm
```

For brevity I omit the list here, but among the contents you can easily see the `carcano.fancymonitor`'s Python package's files installed beneath the system-wide Python site-packages directory. Please note also how, as expected, it does not contain any PyTest files.

Let's now install both RPMs as a whole:

```
sudo rpm -ivh \
python3-carcano-fancymonitor-0.1.0-1.el9.noarch.rpm \
carcano-fancymonitor-0.1.0-1.el9.noarch.rpm
```

This time the installation succeeded.

```
Verifying...          ################################# [100%]
Preparing...          ################################# [100%]
Updating / installing...
   1:python3-carcano-fancymonitor-0.1.################# [ 50%]
   2:carcano-fancymonitor-0.1.0-1.el9 ################# [100%]
```

We can now verify that the Fancymonitor is available as a system-wide package.

Just run

```
fancymonitor -c /etc/fancymonitor/config.xml
```

You can of course repeat the command using the `config.json` and `config.yml` configuration files.

Since they are now installed, when running `rpm` statements, we stop referring to their filename and start using their package name. For example, we can verify they are installed and get information by running

```
rpm -qi python3-carcano-fancymonitor carcano-fancymonitor
```

As you can see, among the information this time there is also the installation date.

Another useful statement is querying the RPM database to show the package configuration files.

```
rpm -qc carcano-fancymonitor
```

The output is as follows:

```
/etc/fancymonitor/config.json
/etc/fancymonitor/config.xml
```

```
/etc/fancymonitor/config.yml
/etc/fancymonitor/logging.conf
/etc/fancymonitor/schemas/probes.dtd
/etc/fancymonitor/schemas/probes.xsd
/etc/fancymonitor/xslt/config.xsl
```

Once finished, let's uninstall the RPM packages by running

```
sudo rpm -e python3-carcano-fancymonitor carcano-fancymonitor
```

Summary

This chapter explained the benefits in terms of improved system maintainability and security provided by packaging applications using system-specific packages so as to be able to install them using the system's package manager.

We also delved into the details of RPM packaging, not only illustrating the SPEC file, but also practicing an example RPM build process. Moreover, we also used the generated RPM files to run some example statements for querying metadata, contents, and requirements of the generated files.

These skills you got in this chapter are crucial for DevOps and DevSecOps professionals: mastering RPM package creation enables seamless deployment on Red Hat family systems, ensuring that applications are consistently delivered in a reliable and efficient manner. This skill not only streamlines the deployment process but also enhances security by allowing for controlled and repeatable installations that better integrate with the target system, leading to more robust and secure applications.

CHAPTER 18

Setup Git and the Pre-commit Framework

In this chapter, we start using the Git Version Control System (VCS) a little more: after quickly introducing how to set it up, we anticipate the explanation of an important feature: the pre-commit hook. The aim is just having a short walkthrough on how to exploit this hook to trigger scripts typically aimed at running compliance checks, aborting the commit process in case of violations detected. We instead postpone thoroughly delving into Git to Chapter 35, which is entirely dedicated to it.

Configure Git

Since in a few of the next labs we will commit changes in the Fancymonitor Git repository, before continuing, we must perform at least the most basic configuration of Git.

When working with it and making changes to versioned content, it is indeed mandatory to configure the Author identity. This information is essential for tracking contributions, attributing work correctly, and facilitating collaboration among multiple developers.

Author identity can be set globally (at the user level) or at the repository level (so overriding the global one as necessary).

If you have not already done it, you must now configure your Git Author identity as follows:

```
git config --global user.email "marco.carcano@carcano.corp"
git config --global user.name "Marco Antonio Carcano"
```


© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_18

You may of course want to adapt the above values to fit your needs: since we specified the `--global` command-line option, the above statement sets the Author identity globally.

Git Commits

While working with any VCS, such as Git, committing is the action of making changes permanently added to the codebase.

This involves also adding a descriptive log (the commit message) that is pushed to the commit history. Along with the commit message, the commit tracks also who made the commit.

So, without Author information, commits would lack clear authorship, making it difficult to manage code reviews, resolve conflicts, or audit changes.

Commits' Compliance

From the perspective of the project lifecycle, committing changes is a critical juncture: each commit adds a consistent point to the project's history, ideally introducing stable logic for new features or bug fixes.

Best practices dictate that code should be sufficiently refined before committing. This means it must adhere to the organization's code-compliance standards, which encompass everything from code style to the necessary information and tags in commit messages.

Relying on individuals to remember to run compliance check scanners before committing is just unrealistic: human nature leads to distractions and forgetfulness. Therefore, it is more effective to implement automated triggers that initiate these checks during specific events.

When source code is versioned using a VCS such as Git, leveraging pre-commit hooks to automatically execute these checks at each commit is advantageous, since this approach ensures that compliance standards are consistently met, reducing the risk of errors and enhancing overall code quality.

Note Pre-commit hooks are entry points that execute before a commit in a Version Control System like Git, allowing for the execution of custom scripts. If the custom script returns an error, the commit operation is aborted, preventing potentially problematic code from being committed. Pre-commit hooks can be employed for various purposes, ranging from launching static code analysis tools to adding notes about the commit to a ticketing system, among others.

The Pre-commit Hook Python Package

While it is possible to manually configure Git's pre-commit hook, it is much more convenient to use off-the-shelf tools such as the `pre-commit` Python package, which is a framework for managing and maintaining multi-language pre-commit hooks.

Since commits are just incremental changes from the previous one, triggering scans at commit time, compared to running manual full scans, has the advantage of reducing waits, saving time compared to running checks on the entire codebase.

Installing the Pre-commit Python Package

To install the `pre-commit` framework, you can use the following command:

```
python3 -m pip install --user pre-commit
```

The `--user` option tells `pip` to install the module in the current user's Python site-packages directory. This is considered best practice, as it helps avoid interfering with the system-wide Python packages provided by the system's package manager. This spares us from the risk of installing potentially conflicting packages that have not gone through the same rigorous testing process.

Configuring the Pre-commit Python Package

The `pre-commit` framework comes with a large repository of pre-built hooks for common tasks, such as code formatting (e.g., `Black`, `Prettier`), linting (e.g., `Flake8`, `ESLint`), and security checks (e.g., `Bandit`).

Note Once the `pre-commit` Python module is installed, you must enable it on every single Git repository where you want to use it. Just change to the root directory of the Git repository and run `pre-commit install`.

Pre-commit reads its configuration from the `.pre-commit-config.yaml` file in the Git repository's root directory.

Its minimal configuration just contains the following empty dictionary:

```
repos: []
```

To help you quickly familiarize yourself with the tool, the Fancymonitor project provides a pre-configured `.pre-commit-config.yaml` file.

To have a look at it, change to the Fancymonitor project root directory:

```
cd ~/devsecops-pillars/fancymonitor
```

Then, you can view its current settings by running

```
cat .pre-commit-config.yaml
```

Upgrading the Scanners

Since as time passes by, new scanner versions are published, the best practice is to periodically upgrade the configured scanners.

First, check out the `main` branch:

```
git checkout main
```

To launch the upgrade process, simply run

```
pre-commit autoupdate
```

This command causes `pre-commit` to automatically check for new tags and update the configuration as necessary.

Note The `autoupdate` command just checks the Git tag of the component to update and updates it to the most current one. This may break compatibility with the system's Python version. For example, recent versions of `pylint` require Python 3.10 and so are not compatible with the Red Hat 9 family distros. In such a scenario, right after running `pre-commit autoupdate`, you must manually reset `pylint` version `v3.3.9` in the `.pre-commit-config.yaml` file. The same applies to `bandit`: the latest version supporting Python 3.9 is `1.8.3`.

If updates are found, after fixing the `pylint` and `bandit` versions as described in the note above, we must commit the changes to Git by running:

```
git commit .pre-commit-config.yaml \
-m "updated the pre-commit framework's repos"
```

Note We updated the pre-commit repositories only in the main branch; if you are working in other branches, you must repeat the above process for each specific branch.

While it is commonly used with Python, `pre-commit` supports hooks written in various languages, allowing you to use it in diverse projects.

Reset the Branch

To avoid problems with the next labs that may be caused by any modification you made while experimenting with this chapter, before continuing please reset the workspace as it originally was by running

```
git reset --hard
```

Summary

In this micro-chapter, we introduced the `pre-commit` framework to automatically trigger actions during the commit process, aborting the commit if any issues arise.

This is just a very short overview aimed at only describing its basic setup: we will see some examples with the actual configuration of its plugins in the next chapters.

Knowing how to deal with the `pre-commit` framework enables the programmatic execution of scripts that can be used for running security scans and compliance checks – we will see examples in the next two chapters.

The ability to perform automatic code scans is crucial for DevOps and DevSecOps professionals, allowing teams to enforce coding standards and security practices consistently across their codebases. This proactive approach not only enhances code quality but also fosters a culture of security awareness among developers, which is one of the expected contributions to be brought by DevSecOps professionals: as DevSecOps professionals, it is our responsibility to manage the automation of security and compliance processes indeed.

For these reasons, understanding how to leverage hooks and the `pre-commit` framework is a must-have.

CHAPTER 19

Auditing Source Code Compliance

This chapter addresses a critical aspect of DevOps and DevSecOps: source code compliance. Source code compliance involves ensuring that code adheres to established standards, regulations, and best practices, which is vital for maintaining quality and security throughout the software development lifecycle.

Neglecting to perform source code compliance checks can result in significant maintainability challenges, particularly in large projects where complexity can escalate quickly.

As DevOps and DevSecOps professionals, it is our responsibility to ensure that projects consistently align with corporate compliance standards. This commitment not only promotes maintainability but also integrates security practices into the development process, ultimately enhancing the overall integrity of the software.

Static Code Analysis Tools

Code quality and maintainability are paramount: as a project grows in complexity, the likelihood of introducing bugs, inconsistencies, and technical debt increases.

The static code analysis helps to cope with this: it is a powerful technique that helps developers identify potential issues in their code without even needing to execute it: by analyzing the source code, static analysis tools provide valuable insights into code quality, such as the adherence to coding standards and potential errors. All of this leads to more robust and maintainable software.

When dealing with Python, two of the most widely used static code analysis tools are `Flake8` and `Pylint`: despite both serving the same purpose, they work in slightly different ways and with varying levels of detail.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_19

By integrating these tools into your workflow, you can foster a culture of quality and continuous improvement, ultimately leading to more successful and maintainable Python projects.

Whether you choose `Flake8` for its simplicity or `Pylint` for its comprehensive analysis, both tools are invaluable assets in the quest for high-quality code.

Flake8

Flake8 is a lightweight tool focused on enforcing a consistent coding style and catching simple errors early in the development process. It combines several other tools under one umbrella, including `PyFlakes`, `pycodestyle`, and the `McCabe` complexity checker.

Flake8 comes with a preset of rules that can be easily customized to meet each project's needs.

The provided benefits are

- **Identify style violations**: It ensures that the code adheres to the `PEP 8` style guide (the de facto standard for Python code formatting). The style rules concern line length, indentation, whitespace usage, and naming conventions.
- **Identify logical errors**: It can detect common programming errors, such as undefined variables and unused imports.
- **Monitor code complexity**: It identifies functions and methods that may be too complex (and so hard to understand and maintain), so that you can rework them, reducing their complexity, for example, splitting the code into multiple functions.

The `.pre-commit-config.yaml` file in the Fancymonitor project root directory contains an example of Flake8 configuration. The relevant snippet is

```
repos:
  - repo: https://github.com/PyCQA/flake8
    rev: 7.3.0
    hooks:
      - id: flake8
```

Flake8 configuration can be tailored by configuring the `[flake8]` stanza in the `setup.cfg` file.

For example, it is possible to add the `ignore` directive with the list of checks to ignore. This way Flake8 focuses only on the issues that matter most to your team, allowing for flexibility in coding style or accommodating existing codebases that may not fully adhere to `PEP 8`.

In the Fancymonitor project, the `[flake8]` stanza is as follows:

```
[flake8]
ignore = E125,E128,E501,F405
```

So it ignores

- `E125`: Continuation line with the same indent as the next logical line
- `E128`: Continuation line under-indented for visual indent
- `E501`: Line too long (more than 79 characters)
- `F405`: Name may be undefined or defined from star imports

Note Since in our setup `Flake8` is managed by the `pre-commit` framework, scans are automatically triggered only at commit time. You may anyway force a run of Flake8 on all the checked-out files of the repository by running `pre-commit run flake8 --all-files`. If you want to see `Flake8` in action when detecting problems, just remove some of the ignored checks from the `flake8` stanza of the `setup.cfg` file and re-run the pre-commit framework with the `--all-files` option as just described.

Pylint

Despite still being a static code analysis tool, Pylint goes beyond style enforcement to analyze code quality, detecting potential bugs and suggesting improvements.

Besides style checks such as identifying inconsistent naming conventions, by performing a thorough analysis of the code, Pylint not only detects errors but checks also for code smells and potential issues such as unreachable code, duplicate code, and so on.

At the end of the run, Pylint generates detailed reports that include metrics such as code complexity, code coverage, and the number of lines of code. This information helps teams to assess the overall health of their codebase and make informed decisions about refactoring and improvements.

In the same way as when using Flake8, developers can disable specific checks, tailoring the scan to be compliant with the specific project or corporate policy.

Compared to Flake8, Pylint may require more configuration and can produce a larger volume of output. However, its depth of analysis makes it invaluable for teams aiming to maintain high code quality and catch subtle issues that might otherwise go unnoticed.

Note Pylint can be easily integrated into popular IDEs and text editors, providing real-time feedback as developers write code. Pycharm, for example, has Pylint enabled out of the box.

You can find an example of Pylint configuration in the `.pre-commit-config.yaml` file stored in the Fancymonitor project root directory. The relevant snippet is

```
repos:
  - repo: https://github.com/PyCQA/pylint
    rev: v3.3.8
    hooks:
      - id: pylint
```

Pylint configuration can be tailored by configuring the`[pylint]` stanza in the `setup.cfg` file.

For example, it is possible to add the `disable` directive with the list of checks to ignore.

In the Fancymonitor project, the `[pylint]` stanza is as follows:

```
[pylint]
disable = wildcard-import, unused-import, missing-module-docstring,
line-too-long, invalid-name, missing-function-docstring, redefined-
outer-name, consider-using-f-string, consider-using-sys-exit, logging-
format-interpolation, broad-exception-caught, unused-wildcard-import,
import-error, too-many-locals, too-many-instance-attributes,
```

```
too-many-arguments, too-many-positional-arguments, redefined-builtin,
unused-variable, missing-class-docstring, broad-exception-raised,
similarities, wrong-import-order, too-many-nested-blocks, arguments-
renamed, consider-merging-isinstance, dangerous-default-value, too-many-
boolean-expressions, unspecified-encoding, undefined-variable, redundant-u-
string-prefix
```

Note Since in our setup `PyLint` is managed by the `pre-commit` framework, scans are automatically triggered only at commit time. You may anyway force a run of `PyLint` on all the checked-out files of the repository by running `pre-commit run pylint --all-files`. If you want to see `Pylint` in action when detecting problems, just remove some of the disabled checks from the `pylint` stanza of the `setup.cfg` file and re-run the pre-commit framework with the `--all-files` option as just described.

To avoid problems with the next labs that may be caused by any modification you made while experimenting with this chapter, before continuing please reset the workspace as it originally was by running

```
git reset --hard
```

Summary

In this short chapter, we explained the importance of source code compliance: we learnt how to use and configure tools like Flake8 and Pylint to perform static code analysis, helping to identify and rectify potential issues in our code.

The ability to perform automatic code scans is crucial for DevOps and DevSecOps professionals, since it allows them to enforce coding standards and security practices consistently.

The outcome of the automated compliance and security scans enables us to quickly check adherence to coding standards and identify potential issues early in the development cycle, reducing the risk of vulnerabilities and bugs in production.

By catching issues before they reach production, we reduce the risk of introducing vulnerabilities and enhance maintainability, ultimately leading to more secure and reliable software.

CHAPTER 20

Scanning Vulnerabilities

This chapter focuses on another critical topic within the DevOps and DevSecOps scope: Source Code Vulnerability Scanning. This is the process of identifying and assessing security weaknesses in the code and its dependencies, allowing teams to address potential threats before they can be exploited.

Neglecting it can lead to significant security issues, particularly in large projects, and may result in the delivery of insecure software.

In the worst-case scenario, such software could serve as a Trojan horse, providing unauthorized access to the systems on which it operates.

As DevOps and DevSecOps professionals, it is our responsibility to promote security by integrating Source Code Vulnerability Scanning into development pipelines.

By doing so, we not only enhance the security posture of our applications, but also foster a culture of accountability and continuous improvement within our teams.

Vulnerability Scanning Tools

Writing software has never been as tricky as nowadays: as technology evolves, so do the tactics of malicious actors, so it is now essential for developers to be proactive in identifying and mitigating vulnerabilities.

Applications often handle sensitive information, such as personal data, financial records, or proprietary business information. These must be carefully protected, avoiding unauthorized access and breaches not only for the customer's own safety, but also to avoid having the organization's reputation damaged and in general an erosion of the customer's trust.

In addition to that, many industries are subject to regulations that require organizations to implement security best practices: failing to identify and address vulnerabilities can not only lead to non-compliance but also result in legal repercussions and financial penalties.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_20

Tools such as `Bandit` and `Safety` provide a huge help in identifying security vulnerabilities in both the source code and its dependencies.

Bandit

Bandit is a security linter specifically designed for Python code: by analyzing the abstract syntax tree (AST) of the source code, it identifies security issues. It does not find only basic issues such as hardcoded passwords: it can also detect insecure coding practices such as using insecure functions (e.g., `eval`, `exec`), improper handling of user input, or insecure file handling and other dangerous coding practices that could lead to vulnerabilities.

Since Bandit is among the list of pre-built hooks supported by `pre-commit`, we can exploit it to have Bandit scans triggered at each commit.

The `.pre-commit-config.yaml` file in the Fancymonitor project root directory contains an example configuration. The relevant snippet is

```
repos:
  - repo: https://github.com/PyCQA/bandit
    rev: 1.8.6
    hooks:
      - id: bandit
```

Bandit configuration can be tailored by configuring the `[bandit]` stanza in the `setup.cfg` file.

The typical need is to add the `skips` directive with the list of checks to ignore.

In the Fancymonitor project, the `[bandit]` stanza is as follows:

```
[bandit]
skips = B506
```

Since Bandit deals with security, we must be very careful before deciding to disable a check. In this specific case, while scanning the Fancymonitor project, Bandit reported the `B506` issue along with the following message:

```
>> Issue: [B506:yaml_load] Use of unsafe yaml load. Allows instantiation of
arbitrary objects. Consider yaml.safe_load().
   Severity: Medium Confidence: High
   CWE: CWE-20 (https://cwe.mitre.org/data/definitions/20.html)
```

```
   More Info: https://bandit.readthedocs.io/en/0.0.0/plugins/b506_yaml_
   load.html
   Location: ./src/carcano/fancymonitor/core/configuration.py:35:15
34    try:
35        return yaml.load(open(manifest_file_path, "rb"),
36            Loader=yamlloader(yaml.SafeLoader))
37    except yaml.YAMLError as e:
```

If you look at the code, it is quite odd that it is marked as affected by this issue: since the function is forcing the use of `yaml.SafeLoader`, then it is supposed to be safe.

After a quick investigation, I stumbled into this issue in the Bandit's bug tracker:

`False yaml.load positive with Loader=*SafeLoader subclass #529 https://github.com/PyCQA/bandit/issues/529.`

The thread ends claiming Bandit just checks for the use of the unsafe `yaml.load` function but cannot detect if it is then forcing the `yaml.SafeLoader` function.

So, after reading that explanation, since Fancymonitor code is forcing the use of `yaml.SafeLoader`, this is a false positive, and so it is safe to skip the checking of the `B506` issue.

Note Since in our setup `Bandit` is managed by the `pre-commit` framework, scans are automatically triggered only at commit time. You may anyway force a run of Bandit on all the checked-out files of the repository by running `pre-commit run bandit --all-files`.

Safety

Note In this section, we will discuss Safety `3.0`, which introduces several changes compared to the previous `2.x` versions. Please note that the information provided here, particularly the instructions for operating using the web UI, may become outdated as new releases of Safety are published.

Conversely from Bandit, which is focused on the code written by you, Safety inspects the Python project's dependencies for known vulnerabilities: it scans the dependency list from the `pyproject.toml` against a database of known security vulnerabilities, alerting if any of the dependencies have known issues.

The report generated by Safety is invaluable for determining which dependency versions are safe to use. It helps developers make informed decisions about which minimum or specific versions to include in the application's requirements, ensuring that only known safe versions are used.

Safety can be easily installed by using pip as follows:

```
python3 -m pip install --user safety
```

Since Safety connects to an online repository of security vulnerabilities, it is necessary to go to https://safetycli.com and register an account. If you have a Google account, you can use your Google credentials to authenticate using OAuth2.

At the time of writing this book, Safety provides both a paid team plan as well as a personal plan (free for private use) limited to 100 scans per month. If you fancy, they also provide a free trial of the team plan.

After registration, you must generate an API key: if you have a team plan, you can generate an organization API key; otherwise you must generate a personal API key.

You will then use the generated API key as the authentication token when running safety scans, setting its value into the `SAFETY_API_KEY` environment variable.

Note To generate the Safety API key, on the top menu of your account area, click on `Organization`. Then, from the left menu bar, click on `API keys`. Make sure to keep these tokens secure, avoiding disclosing them, as they provide access to your Safety account.

This key enables the `safety` command-line tool to access the online vulnerability database.

We can now try to scan the Fancymonitor project – change to its root directory:

```
cd ~/devsecops-pillars/fancymonitor
```

Safety scans are supposed to be run in the feature branch you are working on: this way you can refine your work according to its findings before merging it into the related long-lasting branch (such as the `main` branch).

The only exception is if you have an existing project and you are running Safety for the first time: in this case you must launch this initial run in the long-lasting branches and update them according to its findings, in compliance with your branching policy.

In our scenario, since it is the only long-lasting branch in the Fancymonitor project, we must check out the `main` branch:

```
git checkout main
```

To run the scan, export the access token into the `SAFETY_API_KEY` variable and run the following statement:

```
export SAFETY_API_KEY=foo
safety scan --disable-optional-telemetry
```

The output of my run is as follows:

```
Safety 3.6.0 scanning /home/vagrant/devsecops-pillars/fancymonitor
2025-08-13 01:56:02 UTC

Account: API key used
Project: fancymonitor
 Git branch: main
 Environment: development
 Scan policy: fetched from Safety Platform, ignoring any local Safety CLI
policy files

Python detected. Found 1 Python pyproject.toml file

 src/pyproject.toml: No issues found.


Tested 2 dependencies for security issues using policy fetched from Safety
Platform
1 vulnerability found, 1 ignored due to policy.
0 fixes suggested, resolving 0 vulnerabilities.

Project dashboard: https://platform.safetycli.com/codebases/fancymonitor/
findings?branch=main
```

At the first scan, besides creating the `.safety-project.ini` configuration file in the current working directory, Safety also generates in our online Safety account a dedicated project along with its scan policy.

You can access it by opening in your web browser the Project dashboard's URL provided in the last line of the output.

The above output contains also a message we must investigate:

```
1 vulnerability found, 1 ignored due to policy.
```

The important part is `1 ignored due to policy`, which means the run found a vulnerability, but the policy prevented the scan from failing.

We can change this default behavior by setting `unpinned-requirements` to `false`: this way every vulnerability found during the scan will be reported along with the related remediation.

To achieve this, we must modify the scan policy in the Safety online web UI:

- On the top bar, click on `Codebase`, then select the project we just generated while running the scan.
- Once this project's details are loaded, click the `Codebase settings` button on the top right.
- Once the project settings section is loaded, on the left pane menu, click on `policies` and then click on the only available one to edit it.

The value we need to modify is available only in the Advanced Configuration, so we must click on `Advanced Configuration` at the bottom right.

This will show you the policy contents in JSON format: just set `unpinned-requirements` to `false` in the web UI as shown in the picture below, and save the changes by clicking the `Update policy` button.



Let's now re-run the scan, this time generating a more detailed output, saving it as HTML:

```
safety scan --disable-optional-telemetry \
--detailed-output --save-as html src/log/safety.html
```

This time the scan reveals the details of the vulnerability:

```
 lxml >= 4.6.5 [1 vulnerability found]
  -> Vuln ID 50748: CVE-2022-2309, CVSS Severity HIGH
```

Along with where to find more information on it:

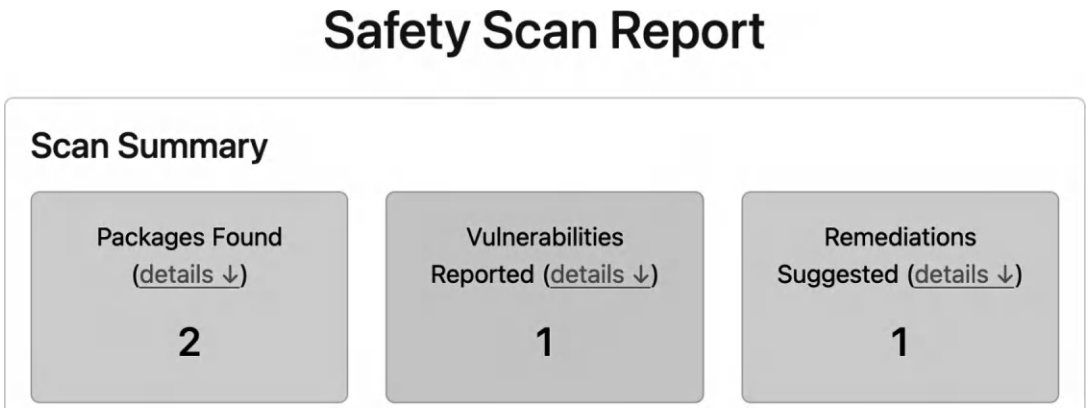
```
Learn more: https://data.safetycli.com/p/pypi/lxml/eda/?spec=%3E%3D4.6.5
```

In addition to that, the scan fails clearly reporting a critical problem was found:

```
Tested 2 dependencies for security issues using policy fetched from Safety
Platform
1 vulnerability found, 0 ignored due to policy.
1 fix suggested, resolving 1 vulnerabilities.
Project dashboard: https://platform.safetycli.com/codebases/fancymonitor/
findings?branch=main
 Scan-failing vulnerabilities were found, returning non-zero exit code: 64
```

Since we provided the `--save-as html src/log/safety.html` option, we can have a look at the `src/log/safety.html` file, opening it with a browser.

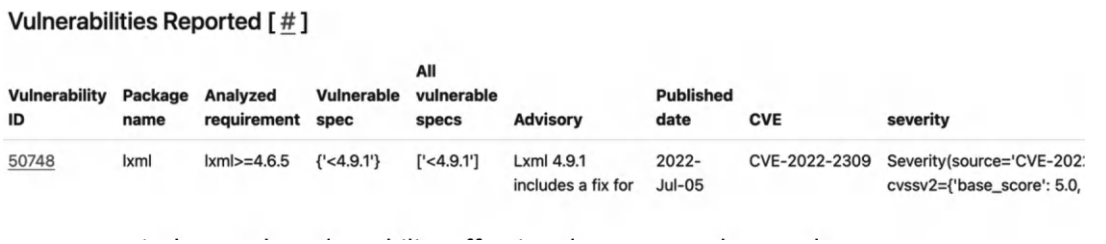
The top of the report shows the following:



As you can see, by checking the contents of the `pyproject.toml` file, Safety detected that Fancymonitor requires two packages and found one vulnerability, suggesting also a fix.

To view the detected vulnerabilities, just click on the `details` link of the `Vulnerabilities Reported`: you will be redirected to the bottom of the report file.

There you will find the list of detected vulnerabilities:



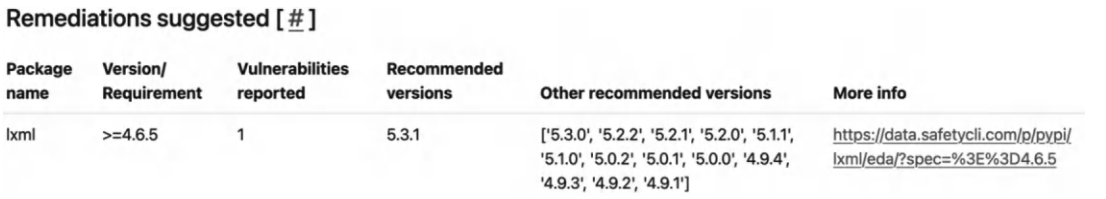
Vulnerabilities Reported [#]

Vulnerability ID	Package name	Analyzed requirement	Vulnerable spec	All vulnerable specs	Advisory	Published date	CVE	severity
50748	lxml	lxml>=4.6.5	{'<4.9.1'}	['<4.9.1']	Lxml 4.9.1 includes a fix for	2022-Jul-05	CVE-2022-2309	Severity(source='CVE-202 cvssv2={'base_score': 5.0,

As you can see, it detected a vulnerability affecting the `lxml` Python package.

To know more about the vulnerability, click on the link of the `Vulnerability ID` `50748` to open the page with the detailed description (https://data.safetycli.com/v/50748/eda).

This reveals that the vulnerability was fixed in `lxml` version `4.9.1`. If you scroll down to the bottom of the report, you will find also the details about the suggested fixing:



Remediations suggested [#]

Package name	Version/ Requirement	Vulnerabilities reported	Recommended versions	Other recommended versions	More info
lxml	>=4.6.5	1	5.3.1	['5.3.0', '5.2.2', '5.2.1', '5.2.0', '5.1.1', '5.1.0', '5.0.2', '5.0.1', '5.0.0', '4.9.4', '4.9.3', '4.9.2', '4.9.1']	https://data.safetycli.com/p/pypi/lxml/eda/?spec=%3E%3D4.6.5

Let's now follow Safety's advice and fix this security vulnerability.

First things first, just to avoid problems that may be caused by any modification you made while experimenting with this chapter, before continuing please reset the workspace as it originally was by running

```
git reset --hard
```

Then, as per best practice, create a dedicated feature branch as follows:

```
git checkout -b f-bump-lxml-requirements
```

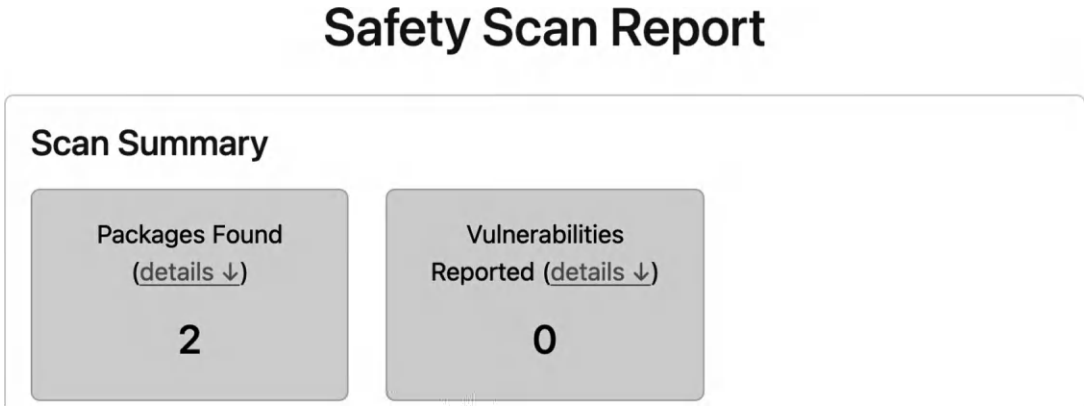
In the `pyproject.toml` file, modify the `dependencies` list in the `project` stanza as suggested, forcing a `lxml` version above or equal to `4.9.1`.

The following snippet shows only the lines relevant to this change:

```
[project]
dependencies = [
    "pyyaml >= 5.4.1",
    "lxml >= 4.9.1"
]
```

To make sure we fixed the vulnerability, we must re-run the scan.

At the time of writing this book, this fix was enough, and the scan didn't report any other vulnerabilities.



We must now make this change permanent but, before proceeding, first, add the file to the staging area:

```
git add src/pyproject.toml
```

then, commit the changes:

```
git commit -m 'raised lxml version requirements to 4.9.1 or above to comply
to the safety CLI report'
```

This operation will trigger the `pre-commit` framework: in this case nothing is really scanned, since the change is related to the project configuration file and so not to source code.

```
[INFO] Installing environment for https://github.com/PyCQA/flake8.
```

```
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
bandit...............................(no files to check)Skipped
flake8...............................(no files to check)Skipped
pylint...............................(no files to check)Skipped
[f-bump-lxml-requirements 0148462] raised lxml version requirements to
4.9.1 to comply to the safetycli report
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Note In a real-life scenario, at this point we would also run `git push -u origin f-bump-lxml-requirements` and raise a pull request using a remote web UI. We will discuss this in detail in Chapter 35, when delving into Git.

Summary

In this chapter, we explained the importance of Source Code Vulnerability Scanning: we conducted vulnerability scans on our code and its dependencies using Bandit and Safety, ensuring that our project is secure from known vulnerabilities.

These skills are vital for DevOps and DevSecOps professionals as they help to proactively identify and mitigate security risks associated with third-party libraries and frameworks.

By regularly scanning for vulnerabilities, potential threats can be addressed before they can be exploited, thereby safeguarding our applications and protecting sensitive data.

Integrating security checks into the version control workflow is one of the cornerstones of DevSecOps, since it ensures that only code that meets security standards is merged into the repository: this not only streamlines the development process, but also fosters accountability among team members, ultimately leading to a more secure and robust software development lifecycle.

PART VII

Security

Within DevSecOps engineering, security does not function as an isolated phase or an external layer applied to a finished product; it constitutes an active, structural property of the infrastructure itself. Frequently, security is relegated to a downstream compliance audit, introducing operational silos between development, operations, and security teams. This section transcends those fragmented models by delivering the core cryptographic and low-level engineering skills required to integrate defense mechanisms directly into the platform architecture. You will examine the mechanics of system hardening—ranging from Linux capabilities and strict access control models to the anatomy of digital trust. This framework bridges the gap between architectural theory and operational execution, leveraging GPG to enforce cryptographic verification across the software supply chain against code tampering, and orchestrating automated Public Key Infrastructures (PKI) via the ACME protocol to govern machine identity. Mastering this section moves past superficial boundary defense to ensure your systems are resilient by design, verifiably authentic, and capable of mitigating internal and external threats systematically.

CHAPTER 21

System Security

It is now time to delve more into the security aspects of DevSecOps: DevSecOps professionals are also tasked with creating and maintaining secure systems, protecting sensitive data, and effectively mitigating risks, all while aligning security practices with development and operations. Achieving these goals requires strong security skills.

In this chapter, we start by examining access control mechanisms, such as POSIX (Portable Operating System Interface) permissions and POSIX ACLs (Access Control Lists), and we highlight how various security-related metadata is stored within extended attributes.

We then move to illustrating privilege escalation techniques, focusing on POSIX special permissions, Sudo, and Linux capabilities.

We also explore the SELinux (Security-Enhanced Linux) framework, demonstrating how its Mandatory Access Control (MAC) model contributes to hardening systems against potential threats.

Finally, we provide an overview of operating cryptography using OpenSSL, illustrating how it can be utilized to secure communications and data.

This chapter includes several example commands you can run to demonstrate the concepts in action. To keep things focused, we'll establish a baseline lab environment upfront, so you don't have to set it up repeatedly for each example.

Let's begin by creating the `practitioner` user account:

```
sudo adduser practitioner
sudo passwd practitioner
```

and the monitoring, sales, and devops groups:

```
sudo groupadd sales
sudo groupadd monitoring
sudo groupadd devops
```


© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_21

Let's then assign the `practitioner` user to the `devops` group:

```
sudo usermod -a -G devops practitioner
```

We must also assign the current user to the `devops` group:

```
sudo usermod -a -G devops $(whoami)
```

Note Even though the command above assigns your current user to the `devops` group, this change only takes effect in new sessions. To apply it immediately, log out of your current session (type `exit` or press CTRL+D) and log back in.

Since in a few examples we will need a text file, let's create the `/tmp/foo.txt` containing 1000 random words. This can be easily achieved by using `shuf` application feeding it with the `/usr/share/dict/words`. Since this file is provided by the `words` RPM package, if it's not already installed, we must first install it as follows:

```
sudo dnf install -y words
```

we can now create the `/tmp/foo.txt` by running

```
shuf -n 1000 /usr/share/dict/words | tr '\n' ' ' > /tmp/foo.txt
```

We must then create a few directories for the examples for practicing with managing permissions:

```
sudo mkdir -m 755 /srv/nfs /srv/nfs/exports \
/srv/nfs/exports/{shared,devops,sales,orders}
```

Lastly, we need to mock an application to perform a few security-related settings – we achieve this by copying the `whoami` command-line utility as follows:

```
sudo mkdir -m 755 /opt/secureapp /opt/secureapp/bin
sudo cp -p /usr/bin/whoami /opt/secureapp/bin
sudo chown practitioner: /opt/secureapp/bin/whoami
```

Note Keep in mind that this baseline setup will serve as the starting point for examples not only in this chapter, but also in the next two chapters. To avoid disrupting the environment, please run only the commands and instructions provided in this book – do not execute any additional commands or make other changes. This ensures consistency and prevents issues with the examples.

Files And Directories

In Linux everything is a file

This sentence is often attributed to the philosophy of Unix-like operating systems, including Linux. While Andrew S. Tanenbaum, a prominent computer scientist, has discussed similar concepts, the exact origin of the phrase is difficult to pinpoint, as it has become a widely accepted adage in the Linux community over time.

This is a key concept to understand: in Linux, directories, symbolic links, device files, sockets, and FIFOs (named pipes) are all treated as files. This approach provides a consistent interface for interacting with various system resources, making it essential for users and administrators to have strong skills in file security.

In Linux, each file object has a wealth of metadata that stores different attributes. Some of this metadata is automatically managed by the Virtual File System (VFS) layer, while others can be set for specific security purposes or advanced data classification.

Understanding this metadata is mandatory for implementing effective security measures, managing permissions, and ensuring proper access control in a Linux environment.

Date and Time Attributes

From the activity tracking perspective, it is easy to deduce the importance of these attributes.

Table 21-1. *Date and time attributes*

Attribute	Visual Representation	Access Type
atime	Last access time	The last time the file was accessed for reading
mtime	Last modification time	The last time the file's content was modified
ctime	Last status change time	The last time the file's content or metadata changed
crtime	Creation time	The time when the file was created – mind that not every filesystem implements this attribute. For example, it is supported by `ext4`, `XFS`, `Btrfs`, `JFS`, and `F2FS`

These attributes can be easily retrieved using the `stat` command-line utility. For example:

```
stat /etc/passwd
```

Note In addition to date and time attributes, the `stat` command provides several other useful details about a file, such as the file type (e.g., regular file, directory, or symbolic link), the `inode` number (a unique identifier for the file's inode on the filesystem), the block size (the size of each block used by the filesystem), the number of blocks (the total blocks allocated for the file's data), the POSIX permissions (the access rights for owner, group, and others), the user ID (the numeric ID of the file's owner), the group ID (the numeric ID of the owning group), and the link count (the number of directory entries pointing to the `inode`).

POSIX Standard Permissions

POSIX (Portable Operating System Interface) is an open standard that defines a set of interfaces for software compatibility with variants of Unix and other operating systems, ensuring portability across different platforms.

Beneath its specifications, it also claims POSIX permissions, which are a set of access

control mechanisms used in Unix-like operating systems to manage the access level of specific subjects to files and directories.

Table 21-2 summarizes the access levels defined by the POSIX standard implemented in Linux.

Table 21-2. *POSIX basic access levels*

Numeric Permission	Symbolic Permission	Visual Representation	Basic Access Level
0		`---`	`No Access`
1	`x`	`--x`	`Execute` when dealing with files, or `metadata access` and `directory traverse` when dealing with directories
2	`w`	`-w-`	`Write`
4	`r`	`r--`	`Read`

These basic access levels can be combined as necessary: for example, the symbolic `rw` permission corresponds to `6` (4+2) numeric permission, `rx` to `5` (4 + 1), and so on.

POSIX standard permissions' subjects are

- `User`: The user owning the file or directory
- `Group`: The group owning the file or directory
- `Others`: Any user not being the owner and not being of the owning group of the file or directory

POSIX permissions are set using the `chmod` command-line utility.

Note The owner of an object can always run `chmod` on it without needing administrative rights.

For example, since our current user is the owner of the `/tmp/foo.txt` file, we can run the following statement, using symbolic permissions, to assign `Read/Write` access to the owning user, `Read/Write` access to the owning group, and `Read` access to the others:

```
chmod u=rw,g=rw,o=r /tmp/foo.txt
```

Let's verify it using the `ls` command-line utility:

```
ls -l /tmp/foo.txt
```

The output is

```
-rw-rw-r--. 1 vagrant vagrant 10195 Feb 24 23:21 /tmp/foo.txt
```

The first part of the line contains the visual representation of the POSIX permissions, grouped by the subjects `User` (`rw-`), `Group` (`rw-`), and `Others` (`r--`): the meaning of these subjects is as described by the previous table.

The previous `chmod` statement can be also written using numeric permissions as follows:

```
chmod 664 /tmp/foo.txt
```

Note Besides the subjects `u` (user), `g` (group), and `o` (others), there is also the `a` subject, which means "all." It is used to make a change that affects every subject. For example: `chmod a-xw /tmp/foo.txt`.

It is possible to alter the current permissions set by using symbolic permissions preceded by the `+` character to add permissions or by the `-` character to remove permissions.

For example:

```
chmod u+w,g+w,o-r /tmp/foo.txt
```

Using the current permissions of the `/tmp/foo.txt` file as baseline, it adds `Write` permission to both the `User` and `Group` owners and removes `read` permissions from the `Others`.

The owning `User` is set by using the `chown` command-line tool, whereas the owning `Group` is set by using `chgrp`.

For example, to assign the `practitioner` user ownership of the `/srv/nfs/exports/devops` directory, just run

```
chown practitioner /srv/nfs/exports/devops
```

The outcome is a permission denied error: this is because running `chown` always requires administrative rights.

Let's re-run the statement, this time using the sudo wrapper:

```
sudo chown practitioner /srv/nfs/exports/devops
```

If we check now the outcome by running

```
ls -dl /srv/nfs/exports/devops
```

We see that this time the statement succeeded.

Note When using chown, it is possible to assign the user along with its primary group using a single statement by appending the colon character to the specified user. For example: `sudo chown practitioner: /srv/nfs/exports/devops`.

Let's now set the current user as owner of the /srv/nfs/exports/devops directory:

```
sudo chown $(whoami) /srv/nfs/exports/devops
```

If we now check the current permissions of the /srv/nfs/exports/devops directory by running

```
ls -dl /srv/nfs/exports/devops
```

we get the following output:

```
drwxr-xr-x. 2 vagrant practitioner 6 Feb 24 23:20 /srv/nfs/exports/devops
```

Since I'm logged in as the vagrant user, the above output reveals that the user I'm using is the owner of the /srv/nfs/exports/devops directory, with full access (rwx) to it.

Note Conversely from chown, which always requires administrative rights, an unprivileged user is eligible to run chgrp, but only on objects to which he has write access to and only if he is member of the target group.

To test in practice the above note, let's check the groups the current user belongs to

```
groups
```

The output is

```
vagrant build devops
```

Since the `devops` group is listed in the output, then we have all the requirements to run a `chgrp` statement to assign to the `devops` group the ownership of the `/srv/nfs/exports/devops` directory without the need to use `sudo`.

Let's try it as follows:

```
chgrp devops /srv/nfs/exports/devops
```

As you can see, as confirmation of the previous note, we were able to run it without administrative privileges.

POSIX Special Permissions

The POSIX standard also reserves an additional byte for special permissions, often referred to as special permission bits.

These can be expressed using either numeric or symbolic notation, as summarized in Table 21-3.

Table 21-3. *POSIX special permissions*

Numeric	Symbol	Bit Name	Granted Permission
0		Not Set	The numeric permission `0` is used to show that there are no POSIX special permissions set. As an example, if `/tmp/foo.txt` is the target, we can remove special permissions by running `sudo chmod a-st /tmp/foo.txt`.

(*continued*)

Table 21-3. (*continued*)

Numeric	Symbol	Bit Name	Granted Permission

1	t	`sticky`	The numeric permission `1` is used to indicate that the `sticky` bit is set, or to set it, resetting the existing POSIX special permissions. The `sticky` bit restricts the ability to delete or rename files within that directory. Specifically, it ensures that only the owner of a file (or the `root` user) can delete or rename that file, even if other users have write permissions on the directory.
			This is particularly useful in shared directories, such as `/tmp` and `/var/tmp`, where multiple users may create temporary files. When set, the sticky bit prevents accidental or malicious deletions by ensuring users can only manage their own files.
			To set the sticky bit on an existing directory, use the symbolic mode with `chmod +t`. For example, to add the sticky bit to the `/srv/nfs/exports/shared` directory:
			`sudo chmod +t \`
			`/srv/nfs/exports/shared`

2	s	`setgid`	The numeric permission `2` is used to indicate that the `setgid` bit is set, or to set it, resetting the existing POSIX special permissions. When the `setgid` bit is set on a directory, any new files or subdirectories created within it inherit the group ownership of the directory itself, rather than the primary group of the user who created them. This facilitates collaboration in shared environments, such as network shares, by ensuring new objects are accessible to all members of the group.
			The `setgid` bit can also be applied to executable files. In that case, the program runs with the permissions of the group that owns the file, rather than the user's primary group.
			To set the `setgid` bit on an existing directory, use the symbolic mode with `chmod g+s`. For example, to add the `setgid` bit to the `/srv/nfs/exports/devops` directory:
			`sudo chmod g+s \`
			`/srv/nfs/exports/devops`

(*continued*)

Table 21-3. (*continued*)

Numeric	Symbol	Bit Name	Granted Permission

4	s	`setuid`	The numeric permission `4` is used to indicate that the `setuid` bit is set, or to set it, resetting the existing POSIX special permissions. When the `setuid` bit is set on an executable binary file (not a script), the operating system runs the process with the privileges of the user who owns the file, rather than the user who launched it. This allows users to perform tasks requiring higher privileges. A classic example is the `passwd` command. Password files contain sensitive information and are secured against unauthorized access. To let regular users change their own passwords, the `passwd` executable is owned by `root` and has the `setuid` bit set, enabling users to run it with root privileges to modify their password entries. To set the `setuid` bit on an existing binary file, use the symbolic mode with `chmod u+s`. For example, to add the `setuid` bit to `/opt/secureapp/bin/whoami`, you would use the following command:
			`sudo chmod u+s \`
			`/opt/secureapp/bin/whoami`
			In this example, no matter which user runs `/opt/secureapp/bin/whoami`, the output will always reflect the ownership of `practitioner` due to the elevated privileges.

Note `setuid` and `setgid` programs can be very dangerous if they are not designed with security in mind. For example, consider a `setuid` application that provides an option to specify the path of the file to process from the command line. If not properly validated, this can be exploited to access every file on the system, potentially leading to unauthorized access to sensitive information. For this reason, it is crucial to carefully review and test applications before granting them `setuid` or `setgid` permissions. Ensuring that these applications are secure and properly handle user input can help mitigate the risks associated with elevated privileges.

You can set special permissions using numeric notation. In this case, specify the special permission digit first, followed by the three digits representing the standard POSIX permissions.

For example, to set the `setgid` bit on the `/srv/nfs/exports/devops` directory while granting `read`, `write`, and `execute` (`traverse`) access to the owner `User` and `Group` and denying access to everyone else, you would run the following command:

```
sudo chmod 2770 /srv/nfs/exports/devops
```

Extended Attributes

Extended attributes are additional information stored in a separate area linked to the inode. Using a separate area enables overcoming the constraint of fitting the storage space of the inode, allowing to store additional metadata for security and file classification purposes.

They are managed using the `getfattr` and `setfattr` command-line utilities:

- `getfattr`, invoked with the `-d` option, dumps all extended attributes along with their values for a specified file. If invoked with the `-n` option followed by the name of an extended attribute, it displays the value of that specific attribute.
- `setfattr`, if invoked with the `-n` option, sets the desired extended attribute with the provided values. If invoked with the `-x` option, it removes the specified extended attribute.

Note Mind `getfattr` and `setfattr` are provided by the `attr` RPM – if it is not already installed, run `sudo dnf install -y attr`.

Table 21-4 summarizes the four available extended attribute types.

Table 21-4. *Extended attribute types*

Type	Prefix	Description
User	`user.`	Used to provide additional user metadata as multiple-purpose tags.
System	`system.`	Used to provide additional system metadata. For example, POSIX ACLs are stored using the `system.posix_acl_access` system extended attribute.
Security	`security.`	Used to provide additional security metadata. For example, SELinux contexts are stored in the `security.linux` security extended attribute.
Trusted	`trusted.`	Used to store integrity information.

Let's illustrate them with a couple of example use cases:

1) Add metadata about the creator and version of the `/tmp/foo.txt` file.

```
setfattr -n user.creator -v "Carcano Marco" /tmp/foo.txt
setfattr -n user.comment -v "version 1" /tmp/foo.txt
```

We can verify the attributes are set by running

```
getfattr -d /tmp/foo.txt
```

2) Remove the `user.comment` extended attribute we just added:

```
setfattr -x user.comment /tmp/foo.txt
```

and check again the outcome by running

```
getfattr -d /tmp/foo.txt
```

As you can see, the `user.comment` attribute has been removed.

As necessary, it is of course possible to dump all the extended attributes of a specific type. The mechanism involves specifying in the dump command the pattern the dumped attribute must match. For example, to dump the `Security` extended attributes

```
getfattr -d -m security /tmp/foo.txt
```

The output is

```
security.selinux="unconfined_u:object_r:user_tmp_t:s0"
```

By observing the above output, if you already worked with SELinux, you certainly now could infer where SELinux stores its tags.

As an example to demonstrate the use of the `trusted` extended attributes, we can see how to use `trusted.sha256sum` to store the `SHA256` checksum of the `/tmp/foo.txt` file.

First, let's calculate and store it in the `CHECKSUM` variable:

```
CHECKSUM=$(sha256sum /tmp/foo.txt | awk '{ print $1 }')
```

Once done, we can store it in the `trusted.sha256sum` extended attribute:

```
sudo setfattr -n "trusted.sha256sum" \
-v "${CHECKSUM}" /tmp/foo.txt
```

We can check the outcome by running

```
sudo getfattr -d -m trusted  /tmp/foo.txt
```

POSIX ACLs

POSIX ACLs provide a more granular permission mechanism than the POSIX standard permissions, since they enable tailoring permissions for multiple users and groups on a single file or directory.

Note ACL can be set only on a filesystem mounted with the `acl` option – which is the default for most of the filesystems.

The basic syntax for adding an ACL entry is as follows:

```
setfacl -m u:username:permissions file_or_directory
setfacl -m g:groupname:permissions file_or_directory
```

As with POSIX standard permissions, permission symbols are `r` for read, `w` for write, and `x` for execute or directory traverse.

We can practice a bit with this running example of applying ACL entries to the `/tmp/foo.txt` file.

As a requirement for these examples, we must first assign ownership to the `root` user and `root` group as follows:

```
sudo chown root: /tmp/foo.txt
```

Then we start by playing with ACLs by granting `rw` permissions to the `practitioner` user:

```
sudo setfacl -m u:practitioner:rw- /tmp/foo.txt
```

If we now check the permissions

```
ls -l /tmp/foo.txt
```

the output is as follows:

```
-rw-rw----+ 1 root root 10173 Feb 14 23:49 /tmp/foo.txt
```

It shows an extra + right after the POSIX permissions: this reveals the presence of a POSIX ACL.

If you change your mind and decide not to trust a `practitioner` any longer, you can revoke every permission from them as follows:

```
sudo setfacl -m u:practitioner:--- /tmp/foo.txt
```

Note The above statement is not deleting the previous ACL entry that granted the `practitioner rw` access: it is overwriting it claiming that the `practitioner` doesn't have any kind of access, or, in other words, we are explicitly denying access.

If you just decide to remove the ACL entry affecting the `practitioner` user, just type

```
sudo setfacl -x u:practitioner /tmp/foo.txt
```

If you need to remove the whole ACL, just use the `-b` option as follows:

```
sudo setfacl -b /tmp/foo.txt
```

Note Even if you remove one by one every ACL entry from the ACL, the whole ACL is not removed: it's just an empty ACL – if you run `ls -l`, the extra + sign is displayed anyway indeed. The only way for removing a whole ACL is using the `-b` option as shown above.

When targets of the `setfacl` statements are directories, it is also possible to add an ACL entry used as the default one for the whole directory tree originating from there:

```
sudo setfacl -d -m g:sales:rwx /srv/nfs/exports/sales
```

Note Default ACLs are not applied to the target directory itself: they are set as default to every newly created children of that directory and to these children's descendants. That means already existent children are not affected by the default ACL.

It is of course possible to remove the default ACL as follows:

```
sudo setfacl -k /srv/nfs/exports/sales
```

If necessary, it is possible to specify also the `ACL mask`: this is a special entry that defines the maximum access permissions that are granted to users and groups for a file or directory.

The mask applies to all user and group entries in the ACL, effectively limiting their effective permissions.

To see ACL masks in action, let's first grant `rwx` access to the `practitioner` user on the `/srv/nfs/exports/sales` directory:

```
sudo setfacl -m u:practitioner:rwx /srv/nfs/exports/sales
```

Then, let's set the `r-x` mask as follows:

```
sudo setfacl -m m:r-x /srv/nfs/exports/sales
```

If you verify it with `getfacl`, now the `practitioner` user's effective permissions are cut to `r-x`.

Note The ACL mask acts as a "ceiling" for permissions. The effective permission for a named user, the owning group, or a named group is the most restrictive (intersection) of their specific ACL entry and the ACL mask. However, the mask does not apply to the file owner or to "others," who always retain their defined permissions regardless of the mask value.

The whole ACL can be dumped using the `getfacl` statement. For example:

```
getfacl /srv/nfs/exports/sales
```

By combining `getfacl` and `setfacl`, it is even possible to copy the whole ACL of an object into another one.

For example:

```
getfacl /srv/nfs/exports/sales | sudo setfacl --set-file=- /srv/nfs/
exports/orders
```

Note One word of caution: when doing backups always make sure your utility supports POSIX ACLs and you are using the option to enable its support as necessary. For example, modern `tar` versions support ACLs, but you must explicitly enable the feature by passing the `--acls` parameter. So, for example, to back up the above exports, you must run `sudo tar cfvz exports.tar.gz --acls /srv/nfs/exports`, whereas to restore it you must run `sudo tar xfpvz exports.tar.gz --acls`.

Mount Options

When mounting filesystems, you can specify mount options to enable or disable features or to alter their default behavior.

Some of these options impact security and are summarized in Table 21-5.

Table 21-5. *Security-related filesystem mount options*

Numeric Permission	Access Type
user_xattrs	Enable extended user attributes support, allowing users to set and retrieve extended attributes on files, which can be used to set additional features
ro	Prevent write access to the filesystem contents: useful to prevent accidental or malicious modifications of data in filesystems with contents that are required to be static
nodev	Limit the attack surface by preventing the creation and use of special device files
nosuid	Limit the attack surface by preventing execution of files with the `setuid` bit set
noexec	Limit the attack surface by preventing the execution of files
acl	Enable POSIX ACL support, providing a way to set up more fine-grained access control rules

A very typical use case is mounting the/`tmp` partition, preventing the running of applications and `setuid` files stored beneath it, as well as forbidding the creation of special device files beneath it.

Note These restrictions by the way are a requirement of several security regulations.

In such a scenario, the options of the `/tmp` filesystem in `/etc/fstab` should be as follows:

```
/dev/mapper/system-tmp  /tmp  xfs  noexec,nosuid,nodev  0  0
```

Note Remember that on modern systems every time you modify the`/etc/fstab` file, you must reload `systemd` by running `systemctl daemon-reload`, and only after this can you remount the filesystem you modified – in this case you should run `mount -o remount /tmp`.

In some circumstances, you may need to temporarily allow exceptions, such as executing applications on a filesystem mounted with the `noexec` option.

For example, some (poorly designed) software often extracts temporary installation or update scripts in the/`tmp` partition and then runs them.

In such a need, we can remount temporarily the `/tmp` with the exec option:

```
mount -o remount,exec /tmp
```

and then, when it is no longer needed, restore the `noexec` option back:

```
mount -o remount,noexec /tmp
```

Note Before softening security, even for a short amount of time, make sure there is no real threat: in the above example, before doing that carefully check the software you are installing, understanding what it is doing when creating and running files from the `/tmp` filesystem.

Sudo

The `sudo` command stands for "superuser do": it is a command-line utility that allows a permitted user to execute a command impersonating another user (the `root` user by default).

The impersonation is managed by the `sudo policies`: these are used to clearly define who can do what impersonating whom on which machine.

Most of the time `sudo` is used to run as the `root` user statements requiring administrative rights, but it is not uncommon to run `sudo` statements impersonating other users.

An example use case is to grant the `monitoring` team the right to start, stop, and restart specific services, just to avoid a pointless escalation for minor problems that can be addressed with a simple service restart.

The Configuration Files

The /etc/sudo.conf file contains settings specific to the sudo application itself, whereas the default sudo policies provided by the Linux distribution are set in the /etc/sudoers file.

Custom sudo rules must instead be saved into individual "include" files in the /etc/sudoers.d directory.

The Wheel Group

To understand how to customize the sudo configuration, a good starting point is looking at the contents of the /etc/sudoers file:

```
sudo cat /etc/sudoers
```

This file, in nearly every Linux distribution, includes the following setting:

```
%wheel    ALL=(ALL)    ALL
```

The meaning of the above line is very easy to infer and shows the basic syntax of a sudo rule, that is,

```
who    where=(run-as)    what
```

The who token can be a username or, if it has a leading %, a group name.

So, the above sudo rule means that any member of the wheel group can impersonate, on any machine, any user and run any command.

Note The default policy provided by almost every Linux distro, unless the connected user is root, requires sudo to prompt for the user's own password before running the statement.

The reason a sudo policy requires specifying which machines the grants apply to is that sudo rules are often stored in a central repository. These rules can be kept in a central files repository, with the files then delivered by datacenter automation tools, or they can be retrieved directly from an LDAP server.

Let's test the previous sudo rule to see it in action. Since it applies to members of the `wheel` group, let's add the `practitioner` user to it:

```
sudo usermod -aG wheel practitioner
```

Now, switch to the `practitioner` user by using the `su` command-line utility:

```
sudo su - practitioner
```

By switching to the `practitioner` user, we are impersonating it.

We can now check the sudo rights that are granted to this user:

```
sudo -l
```

When requested, type the `practitioner`'s user password.

After that, sudo will show the user's grants – the last two lines of the output must look as follows:

```
User practitioner may run the following commands on lab01:
    (ALL) ALL
```

Note If you prefer a more readable way of displaying sudo entries, use `sudo -ll`.

Exit the `practitioner`'s session:

```
exit
```

Then, since we no longer need this, remove the `practitioner` user from the `wheel` group:

```
sudo gpasswd -d practitioner wheel
```

Creating a Simple Sudo Rule

The wheel sudo rule is just a convenient default for enabling users to run statements wrapped by sudo. In real life, since security best practices require granting the least possible privileges, it is instead much more common having to write custom sudo rules tailored to specific operations.

These sudo rules usually meet security policies claimed by the security team. As an example, let's suppose the `monitoring` team raised a request to be granted the right for starting, stopping, and restarting the `chronyd` service without being asked for their own password. The request has been approved by the security team and now it's our turn implementing it.

When dealing with file-based sudo rules, the best practice is to store them in file names recalling their matching criteria, to be able to infer to whom the rule applies by just looking at the file name.

In this example, since the matching condition is being a member of the `monitoring` team, the file is `/etc/sudoers.d/monitoring`. Create it with the following contents:

```
%monitoring ALL=(ALL) NOPASSWD: \
    /usr/bin/systemctl start chronyd, \
    /usr/bin/systemctl stop chronyd, \
    /usr/bin/systemctl restart chronyd
```

There are two differences from the `wheel` group rule we saw before:

- Instead of terminating with the `ALL` token, which enables any command, this rule explicitly lists the three granted commands.
- It contains the `NOPASSWD:` token, which states not to prompt for the user's password before running the command.

Note Be careful when writing `sudo` rules: a syntax error disables the whole `sudo` indeed. If you want to play safe, use `visudo`, which performs a sanity check before saving the file.

Continuing our fictional scenario, let's suppose the `practitioner` user has been moved from the `devops` team to the `monitoring` team. To reflect this organizational change, we must remove him from the `devops` group and then add him to the `monitoring` group:

```
sudo gpasswd -d practitioner devops
sudo usermod -a -G monitoring practitioner
```

We can now test the sudo rule by impersonating the `practitioner` user again:

```
sudo su - practitioner
```

We are now supposed to be able to start, stop, and restart the `chronyd` service at will, without being asked for the `practitioner` user's password.

For example, let's try running

```
sudo systemctl restart chronyd
```

As expected, the statement works, and we are not asked for the password.

Let's exit back to the `root` user's shell:

```
exit
```

Note Always remember to grant only the least possible privileges to achieve only the required tasks: for example, if you need to grant multiple services, list them by name. Don't add entries such as `/usr/bin/systemctl stop *`, since a rule like that grants the right to stop any service: guess the outcome if the user by mistake stops the `sshd` service.

Sudo Rule Syntax

The official sudo manual is available at https://www.sudo.ws/docs/man. It's a vast topic, so this book can't cover everything.

For convenience, here's a brief summary of `sudo` policy syntax:

- `who` can be a username or a group name.
- When `who` is a group name, prefix with `%` (e.g., `%monitoring`).
- `who` can also be a numeric user-id or group-id, prefixed with the `#` character (e.g., `#100` for a user-id, `%#100` for a group-id).
- The host can be the `ALL` word (meaning every host) or a FQDN. When explicitly specifying FQDNs, mind it is possible to match multiple FQDNs using the * wildcard character (e.g., `*.carcano.corp`).

- `run-as` is optional – if omitted it defaults to `root`; otherwise it is granted to be executed as the specified user – e.g., `(practitioner)`. If necessary, you can configure it to run in a context as if the user would be a member of a specific group – e.g., `(:group)`. It is of course possible to even specify both username and group to impersonate – in this case the syntax is `(user:group)`.
- As for statements to run
  - You can list more statements separating them using a comma.
  - You must provide the absolute path of each single command.
  - The wildcard character * must be used with caution, since it can lead to matches wider than expected. In addition to that, mind it stops matching at the first / character in paths: for example, `/usr/bin/*` matches `/usr/bin/whoami` but does not match `/usr/bin/X11/xterm`.
  - For added security against tampering, in addition to the command, you can also specify its checksum. For example, `practitioner ALL=(ALL) sha224:2f82c1d6ed8c61c0e618ea5717e291b1acb7711ddb9390b96f5b90a3 /bin/whoami`.
- It is possible to specify options to alter the behavior or the environment. The most used ones are as follows:
  - `NOPASSWD`: Disables prompting for the current user's password
  - `STENV`: Allows the user to set environment variables from the environment of the user running `sudo` – this is enabled on demand by passing the `-E` option when running the `sudo` statement
  - `requiretty`: Enforces that sudo commands can only be executed from a terminal (TTY)

When dealing with large sudo configurations, the best practice is to configure aliases and then reference them in the sudo rules.

A few examples are

```
User_Alias SYSADMINS = %devops, %dba, !external_advisor
Runas_Alias SUITE01_APPS = foo, bar
Runas_Alias APPS = baz, SUITE01_APPS
Host_Alias PROD = appsrv.carcano.corp, idp.carcano.corp, \
192.168.42.0/24, !192.168.42.10
Cmnd_Alias DB = /opt/postgresql/bin, /opt/mysql/bin/mysql
```

Note Mind how the ! character negates the item and how you can write multiline aliases using the \ character right before going to the newline.

Linux Capabilities

Linux capabilities are a set of distinct privileges that can be assigned to executable files (binaries or scripts) to grant them to spawned processes, independently of whether the running user is an administrative (`root`) user or a regular user

In contrast to `sudo`, which grants the full rights of the impersonated user while limiting the set of authorized actions as necessary, capabilities are targeted at executable files and processes, providing the spawned process with only very specific administrative privileges.

This approach enables more fine-grained control over privilege assignment, but has no control on the user spawning it.

Most Commonly Used Capabilities

It is worth mentioning the most commonly used capabilities, which are summarized in Table 21-6.

Table 21-6. *Most used capabilities*

Capability	Description
CAP_NET_BIND_SERVICE	Allows the spawned process to bind also to ports in the reserved range (below 1024). The default behavior is to forbid this since, especially on Internet-facing servers, it may lead to security concerns, especially when dealing with poorly maintained applications not designed with security in mind.
CAP_SYS_ADMIN	Grants the spawned process a wide range of system administration privileges, such as mounting filesystems, configuring network interfaces, and managing system resources. It's often used by system demons and administrative tools.
CAP_DAC_OVERRIDE	Grants the spawned process the right to bypass file permission checks, allowing a process to read, write, or execute files regardless of their permissions. This is useful for backup tools or in general for system utilities that need to access files across different user permissions.
CAP_KILL	Allows the spawned process to send signals to other processes, including those owned by other users. This is commonly used by monitoring tools and process management utilities.
CAP_SETUID and **CAP_SETGID**	Allow the spawned process to change its user ID (UID) and group ID (GID), respectively. These capabilities are often used in `setuid`/`setgid` programs that need to perform actions as a different user or group.

A straightforward example to illustrate these concepts is the Apache HTTPd server's `suexec` wrapper, which is used to run contents as a different user, in compliance with the principle of least privilege.

To use it in our labs, we must first install the RPM providing Apache HTTPd as follows:

```
sudo dnf install -y httpd
```

Getting Capabilities

We can list the capabilities assigned to an executable file by using the `getcap` command.

For example, to check the capabilities assigned to the `/usr/sbin/suexec` file, run

```
getcap /usr/sbin/suexec
```

The output is

```
/usr/sbin/suexec cap_setgid,cap_setuid=ep
```

The `e` and `p` characters after the equal sign are flags.

There are of course a few other flags, but for most of the real-life use cases, it is enough to know that

- If you need to assign a capability to the spawned process, then you need the `e` flag
- If you need to assign a capability to the spawned process and to all the children it may spawn, then you need both the `e` and `p` flags

Setting Capabilities

Capabilities are managed using the `setcap` command-line utility. The configuration syntax is like the string containing the capability returned by `getcap`. You can use the `+` symbol to add a capability and the `-` symbol to remove a capability.

As an example, suppose you installed the Apache Tomcat application server and you want to provide its executable the capability to spawn processes able to bind to privileged ports.

To accomplish this requirement, the statement you would run is

```
setcap 'cap_net_bind_service=+ep' /usr/sbin/tomcat
```

The above command just illustrates the concept: when dealing with services (such as Tomcat) and Systemd is available, the best practice is to configure the capabilities directly in the Systemd service unit itself using the `AmbientCapabilities` directive. By doing so, capabilities are assigned only for the Tomcat process spawned by Systemd, preventing processes spawned by other users from getting that capability.

Note A Systemd service unit is a configuration file that defines how a service – a background process – should be started, stopped, and managed by the Systemd init system. It specifies parameters such as the executable to run, dependencies, and resource limits. Service unit files have the `.service` extension. Files shipped by the Linux distribution are stored in `/usr/lib/systemd/system`, while custom overrides or additions should be placed in `/etc/systemd/system`.

For example, this is the content of an example Systemd service unit you would create for having the `tomcat` service configured to set the `cap_net_bind_service` capability:

```
[Unit]
Description=Apache Tomcat Web Application Container
After=network.target

[Service]
Type=simple
User=tomcat
Group=tomcat
Environment=JAVA_HOME=/usr/lib/jvm/java-11-openjdk
Environment=CATALINA_HOME=/opt/tomcat
Environment=CATALINA_BASE=/opt/tomcat
ExecStart=/opt/tomcat/bin/startup.sh
ExecStop=/opt/tomcat/bin/shutdown.sh
Restart=on-failure
AmbientCapabilities=CAP_NET_BIND_SERVICE

[Install]
WantedBy=multi-user.target
```

Note You can of course list more capabilities separating them using a comma as necessary.

SELinux

SELinux (Security-Enhanced Linux) is a powerful Linux kernel security module designed to reduce the impact of user errors and administrative mistakes that could otherwise lead to security breaches.

It enforces Mandatory Access Control (MAC), which operates after the standard Discretionary Access Control (DAC) by applying policies with `Type Enforcement` (`TE`) `rules`. These rules specify what actions a subject—such as a user or process—can perform on objects, like files or sockets.

To speed up enforcement, SELinux caches access decisions in the `Vector Cache` (`AVC`) within the kernel.

Any violations are logged in the audit file at `/var/log/audit/audit.log`.

Note For better scalability and make TEs more agnostic, SELinux transitions the actual process into a `TE domain`, so rules apply to the `TE domain` rather than the process directly.

To simplify the identification of `subjects` (such as processes) and `objects` (such as files, sockets, etc.), instead of relying on traditional identifiers like usernames, group names, or filenames, SELinux uses security `labels` (or `contexts`) to tag entities. This means, for example, that files belonging to the same `security context` must be tagged using the same label.

For example, imagine a directory with sensitive corporate data labeled as `corporate_sensitive_t`. A TE rule could restrict access to only authorized subjects. In such a scenario, even if POSIX permissions are mistakenly set to allow access for everyone, SELinux's MAC policy would still block unauthorized access.

SELinux's MAC not only protects from data leaks caused by improperly set permissions, but can even help to mitigate the effects of exploitation of vulnerabilities.

SELinux Policy Sets

Table 21-7 summarizes the three kinds of SELinux policy sets provided by Linux distributions.

Table 21-7. *SELinux policy sets*

Policy	Description
targeted	This is the most common policy used in many distributions. It focuses on protecting specific services (like the web server or database) while allowing other processes to run in a less restricted manner.
minimum	This policy enforces MAC on all processes and files, providing a higher level of security, but requiring more configuration and management.
mls	This policy is used in environments that require multi-level security, where data is classified at different levels, and access is controlled based on these classifications.

SELinux Operational Statuses

From an operational perspective, SELinux has three operational statuses as shown in Table 21-8.

Table 21-8. *SELinux operational status*

Status	Description
enforcing	When it finds a policy violation, SELinux tracks it and enforces it by denying access to resources
permissive	SELinux simply tracks policy violations
disabled	SELinux is disabled

The current SELinux status can be checked by running

```
sestatus
```

The running status can be temporarily changed using the `setenforce` command, for example:

```
sudo setenforce permissive
```

Mind that running `setenforce` just applies a temporary change that will be reverted after the reboot: permanently changing the SELinux status requires modifying the `SELINUX` variable in the `/etc/selinux/config` file.

Note For the sake of completeness, the `SELINUXTYPE` variable in the same file is used to specify which SELinux policy to load at boot time. Avoid playing with this because selecting an incorrect policy can lead to system misconfigurations or even render the system unusable.

For example, to set SELinux in `permissive` mode at boot time, run

```
sudo sed -i 's/^SELINUX=.*/SELINUX=permissive/g' /etc/selinux/config
```

It is of course necessary to restart the system to get this change applied.

Note If you tried the above example, please rework and run the `sed` statement to revert the `SELINUX` variable back to `enforcing` and reboot your system before going on with the next examples.

Switching to the `permissive` status is often used during the process of setting up SELinux custom policies, since it enables tracking false-positive violations yet avoiding denying the related access requests. These tracked false-positive violations are then used to write new SELinux policies, before switching back to `enforcing`.

You may prefer to avoid softening the whole security provided by SELinux by switching everything to `permissive`: in this case, you can switch to `permissive` only specific TE domains.

To see this in action, let's first ensure we are in enforcing mode:

```
sudo setenforce enforcing
```

As an example, let's suppose we need to switch to `permissive` only the `httpd_t` domain – just run

```
sudo semanage permissive -a httpd_t
```

We can then list the domain currently set in `permissive` status by running

```
sudo semanage permissive -l
```

To complete this example, we must remove the `httpd_t` domain from the list of the ones in `permissive` state. Just run

```
sudo semanage permissive -d httpd_t
```

Note Be wary that adding a domain to the `permissive` list is a permanent change.

Listing SELinux AVC Violations

When operating with SELinux, it is often needed to check the security violations it tracked.

AVC violations can be displayed by running

```
sudo aureport --avc -i
```

SELinux Tags

Note Operating the SELinux configuration requires the installation of the `setools-console` RPM package: if it is not already installed, please install it now by running `sudo dnf install -y setools-console`.

As we said, both SELinux's `subjects` and `objects` are identified using tags.

The tags' format is

```
user:role:type:level(sensitivity:category)
```

where

Table 21-9. *SELinux tags tokens*

Tag's Token	Description
user	The SELinux user is an identifier used exclusively by SELinux and has no direct relation to the operating system's user accounts. User tags end with `_u`, for example, `system_u`. You can list all SELinux users with the command `seinfo -u`.
role	This component is part of SELinux's Role-Based Access Control (RBAC) model. Role labels end with `_r`, for example, `object_r`. To view all roles, use `seinfo -r`.
type	This represents the type of restriction: a domain for processes or a type for files. It forms the core of the security context, as SELinux policies regulate interactions between types. Type labels end with `_t`, such as `file_t`. Use `seinfo -t` to display all types.
level	If set to something other than `s0`, this is used in Multi-Level Security (MLS) or Multi-Category Security (MCS) policies for fine-grained control. It includes two additional tags: `sensitivity` and `category`. As mentioned, we won't explore this topic in depth.

Most of the command-line tools use the `-Z` option to show the SELinux context of a subject (such as a process) or an object (such as a file).

For example, to view the SELinux context of a file, you can use the `ls` command with the `-Z` option:

```
ls -Z /etc/passwd
```

This command displays the security context of the specified file.

The output should look like this:

```
-rw-r--r--. root root system_u:object_r:passwd_file_t:s0 /etc/passwd
```

In this output

- `system_u` is the user identity.
- `object_r` is the role.
- `passwd_file_t` is the type.
- `s0` is the sensitivity level.

Alternatively, to view the SELinux context of a running process, you can use the ps command with the -Z option:

```
ps axZ | grep "[/]sshd"
```

This command shows the security context of the sshd process, which is the SSH daemon.

The output will include the SELinux context associated with the running process:

```
system_u:system_r:sshd_t:s0-s0:c0.c1023 2550 ? Ss     0:00 /usr/sbin/
sshd -D -u0...
```

Type Enforcement (TE) Rules

Let's illustrate how a SELinux TE rule gets processed by using an example, such as explaining how processes spawned by the /usr/bin/passwd file are authorized to access the /etc/shadow file.

The security context of /etc/shadow (the file storing the user accounts' encrypted passwords) is

```
system_u:object_r:shadow_t:s0
```

whereas the security context of /usr/bin/passwd (the command-line utility used to manage the user's account passwords) is

```
system_u:object_r:passwd_exec_t:s0
```

When a process is spawned by launching an executable file, the security context of that file transitions to a TE domain having by convention as its name the name of the executable file stripped of the _exec token – in this case the context becomes

```
system_u:object_r:passwd_t:s0
```

With this information, we can look up the SELinux policies to find the TE rule that has passwd_t as the source domain and shadow_t as the target.

The command to run for this lookup is

```
sesearch -A -s passwd_t -t shadow_t
```

This returns the content of the TE rule we were looking for:

```
allow domain file_type:blk_file map; [ domain_can_mmap_files ]:True
allow domain file_type:chr_file map; [ domain_can_mmap_files ]:True
allow domain file_type:file map; [ domain_can_mmap_files ]:True
allow domain file_type:lnk_file map; [ domain_can_mmap_files ]:True
allow passwd_t shadow_t:file { append create getattr ioctl link lock map
open read relabelfrom relabelto rename setattr unlink watch watch_reads
write };
```

SELinux Modules

The SELinux policy is composed of several `modules`, each one containing its own Type Enforcement (TE) rules.

Each `module` is a standalone entity used to define `types`, `domains`, and their entry points, as well as the permissions (grants) within a domain.

Table 21-10 summarizes actions that can be performed on modules:

Table 21-10. *SELinux module actions*

Option	Action	Example
-l	List installed module	`sudo semodule -l`
-i	Install a policy module from a file	`sudo semodule -i \ foomodule.pp`
-d	Disable a module	`sudo semodule -d foomodule`
-e	Enable a module	`sudo semodule -e foomodule`
-r	Remove a module	`sudo semodule -r foomodule`

Note Compiled module file names end with `.pp`: the examples in the above table are about a fictional module called `foomodule`.

Module's Booleans

Some SELinux modules export boolean options that you can tweak to alter the behavior of the default rules.

To get a list of all the available booleans along with their state, run

```
sudo semanage boolean -l
```

As an example of boolean, let's consider `samba_share_nfs`. This boolean controls the interaction between Samba (a software suite that provides file and print services to SMB/CIFS clients) and NFS (Network File System): when enabled, it allows Samba to share files that are located on NFS-mounted filesystems.

We can get the current value of the `samba_share_nfs` boolean using the `getsebool` command as follows:

```
getsebool samba_share_nfs
```

If we want to temporarily alter its value, we can use the `setsebool` command as follows:

```
sudo setsebool samba_share_nfs on
```

Note If you want to permanently change a SELinux boolean, you should add the `-P` option. For example: `sudo setsebool -P samba_share_nfs on`.

The rationale behind the existence of SELinux module booleans is to implement strict security rules by default, in line with security best practices, while also providing an easy and granular way to relax these rules as necessary.

A common real-life use case involves authorizing network connections from the `httpd` domain, which is used by the Apache HTTP Server. By default, outbound network connections are denied preventing potential security leaks.

However, there are legitimate use cases where such connections are necessary, such as when running a WordPress content management system that requires access to an external database engine. In this scenario, it is possible to permanently authorize outbound connections from the `httpd` domain by executing the following command:

```
sudo setsebool -P httpd_can_network_connect on
```

File Context Management

In SELinux terms, the set of tags assigned to files and directories is referred to as the file `context`.

To avoid the need for manually running multiple tagging commands for every single file and directory in the system, SELinux maintains a database of regular expressions that match filesystem paths, along with the corresponding contexts to assign when a match occurs.

This database, known as the `file context database` (`fcontext` for short), is used as a reference to automatically assign contexts when new objects are created or to retag existing objects during system operations, such as when files are restored from backups or when the system is booted.

This database can be displayed by running

```
sudo semanage fcontext -l
```

If necessary, you can of course add rules fitting custom-specific use cases.

To see how to work with file context management, as an example, let's suppose we want to create multiple Samba shares beneath the `/srv/samba` directory.

Let's start by creating the `sales` share:

```
sudo mkdir -m 755 /srv/samba /srv/samba/sales
```

Once done, let's check the SELinux context automatically assigned to the directories:

```
ls -1Zd /srv/samba /srv/samba/sales
```

As you can see, the context is unconfined_u:object_r:var_t:s0.

Since this is not the context expected by SELinux for a Samba share, SELinux will block any access attempt to those directories through Samba.

To address this, we must add a custom rule to the `fcontext` database to set the context to `samba_share_t` for contents matching the `/srv/samba(/.*)?` RegEx.

The statement to run is

```
sudo semanage fcontext -a -t samba_share_t "/srv/samba(/.*)?"
```

If we check again the SELinux context

```
ls -1Zd /srv/samba /srv/samba/sales
```

we find that the context is still `unconfined_u:object_r:var_t:s0`.

This should not be a surprise: as we said, the `fcontext` database provides templates automatically used only when creating new contents.

Since the directories were created before adding the rule we just added, it is necessary to launch a manual relabeling.

We can launch it over the whole `/srv/samba` directory tree as follows:

```
sudo restorecon -rv /srv/samba
```

Let's recheck the SELinux context after the labeling:

```
ls -1Zd /srv/samba /srv/samba/sales
```

As you can see, this time the context of the `/srv/samba/sales` directory is `unconfined_u:object_r:samba_share_t:s0` as expected.

Note Mind that, if necessary, you can modify an existing `fcontext` using the `-m` option or delete it using the `-d` option.

Acquainting to Cryptography

To guarantee data confidentiality (whether on the wire or on disk) or to perform secure identification, cryptography is mandatory.

Therefore, skills in this area are not optional for DevOps and DevSecOps professionals, who must possess at least a basic understanding of the subject and how to apply it effectively.

In this section, we delve into how to use the most used types of cryptography: symmetric and asymmetric.

Symmetric Cryptography

Symmetric cryptography relies on symmetric keys, so only one secret key is used by both parties to perform cryptographic operations, such as encrypting, decrypting, or message code authentication.

Symmetric Key

It is just a key containing random data – as an example, we can easily generate a 256-bit one as follows:

```
openssl rand -hex -out key.bin 32
```

Note We specified 32 since rand uses bytes: 256/8=32.

Encrypting Contents

Symmetric cryptography relies on the most straightforward type of encryption: both parties share the same key to encrypt and decrypt.

This sharing requirement has the huge drawback of needing a secure way for confidentially exchanging that symmetric key between the parties.

Commonly used symmetric encryption algorithms include

- `AES`
- `Blowfish`
- `Camellia`
- `Idea`
- `DES, 3DES`
- `RC4, RC5, RC6`
- `Twofish`

Note Be wary that `DES`, `3DES`, and `RC4` are nowadays considered insecure and so their usage is generally not recommended. As for `Blowfish`, although it was once popular, its 64-bit block size limits its effectiveness, especially for encrypting large files, leading to its classification as less secure.

As an example, let's see how to encrypt a file using `cfb8` mode AES-256 encryption with `openssl`.

Note `cfb8` is only one of the modes of operation supported by the AES algorithm: other supported modes, for example, are `cbc`, `cfb`, `ofc`, and `gcm`.

As per best practices, we use an Initialization Vector (IV) to both encrypt and decrypt data – in this particular use case, we need a hexadecimal IV:

```
openssl rand -out iv.hex -hex 16
```

So far, we have these two files:

- `key.bin` contains the symmetric key.
- `iv.hex` contains the Initialization Vector.

We must now find a secure and confidential channel to share them with the other parties.

As an example of content encryption, we are encrypting the contents of `/etc/issue` file into the new `issue.dat` file:

```
IV=$(cat iv.hex)
KEY=$(cat key.bin)
openssl enc -aes-256-cfb8 -in /etc/issue -pbkdf2 -iv ${IV} \
 -K ${KEY} -out issue.dat
unset KEY IV
```

As you can see, after loading the symmetric key and the IV into variables, we just run the openssl encryption statement, immediately unsetting these variables after it.

Note The `-pbkdf2` option tells `openssl` to generate and use a derived key: this is generated by repeatedly calling a pseudo-random function using a salt.

By using the securely received symmetric key and the Initialization Vector, the other parties can decrypt the contents by just running the same statement, swapping the input and output files, and adding the `-d` option:

```
IV=$(cat iv.hex)
KEY=$(cat key.bin)
openssl enc -aes-256-cfb8 -d -in issue.dat -pbkdf2 -iv ${IV} -K ${KEY} -out
issue.txt
```

An alternate way for encrypting and decrypting contents is, instead of using a pre-generated key, having the private key generated on the fly from a password.

In such a scenario, we must rework the previous example a little bit.

The encryption statement is the same as the previous one – just omit the `-K` parameter:

```
openssl enc -aes-256-cfb8 -in /etc/issue -pbkdf2 -iv ${IV} -out issue.dat
```

When prompted, enter the password you want to use for encrypting the contents.

In this case we must share with the other parties the Initialization Vector file `iv.hex` and of course the password.

The decryption statement becomes

```
IV=$(cat iv.hex)
openssl enc -d -aes-256-cfb8 -in issue.dat -out issue.txt -pbkdf2 -iv ${IV}
```

Asymmetric Cryptography

Asymmetric cryptography relies on an asymmetric pair of keys: a `private` key and a `public` key, which is mathematically related to the private one in a way so that it's very hard to derive the `private` key by knowing only the related `public` key.

Very hard means that it would take hundreds, if not thousands, of years to compute it (considering the computational capabilities at the time you generated the key).

Table 21-11. *Asymmetric key types*

Key	Description
Public	It can be shared openly and is used to encrypt data or verify signatures.
Private	It must be kept confidential and is used to decrypt data that was encrypted with the corresponding `public` key or to create digital signatures.

Asymmetric key pairs can be used for the purposes presented in Table 21-12.

Table 21-12. *Asymmetric key pairs purposes*

Policy	Description
Encrypt Contents	A common use case is confidential communications: it is possible to use a pre-shared `public` key of a party and use it to encrypt a message. Due to the mathematical relation, only the party holding the corresponding `private` key will be able to decrypt the encrypted contents.
Sign Contents	A common use case is systems authentication: the `private` key is used to sign a challenge or message. The recipient can then verify the signature using the pre-shared signer's `public` key. If the signature is valid, the signer's identity is confirmed. A second common use case involves combining `symmetric` and `asymmetric` cryptography: a party generates a hash (or MAC) of some data, then signs that hash using its `private` key. Other parties can verify the signature by using the signer's public key to check the hash, ensuring the data's integrity and authenticity.

Note As you probably guessed, everything works only as long as the `private` key remains protected and confidential in the hands of its owner. If the key is compromised, the entire security mechanism is broken. Additionally, for identification purposes, it is essential that trusted `public` keys are shared and carefully verified beforehand to prevent impersonation or man-in-the-middle attacks.

Table 21-13 lists the most used `asymmetric` encryption algorithms.

Table 21-13. *Common asymmetric encryption algorithms*

Policy	Description
RSA	One of the first public key cryptosystems and widely used for secure data transmission. Key sizes typically range from 2048 to 4096 bits.
DSA	Primarily used for digital signatures rather than encryption. Often used in conjunction with other algorithms for key exchange. It is no longer considered secure nowadays.
ECDSA	A variant of DSA that uses elliptic curve cryptography. Provides the same level of security as RSA with smaller key sizes, making it more efficient.

Since the above algorithms are used both by GPG (GNU Privacy Guard) and X.509 Certificates, we will delve into using them in the next two chapters.

Summary

In this chapter, we explored several key security concepts essential for designing, setting up, and operating secure environments for hosting applications:

- **Filesystem security**: We covered important filesystem attributes, including date and time attributes, extended attributes, POSIX standards, special permissions, and Access Control Lists (ACLs) that enhance file security.
- **Security-related mount options**: We discussed how specific mount options, such as `noexec`, `nosuid`, and `nodev`, can be used to strengthen system security by limiting executable permissions and device file interpretations.
- **Privilege escalation with sudo**: We learned how to use `sudo` for executing commands with elevated privileges and how to configure a secure sudo policy to control access and maintain an audit trail.
- **Linux capabilities**: We examined Linux capabilities, which allow for fine-grained control over process permissions, reducing the risks associated with running processes as root.
- **SELinux overview**: We provided an overview of Security-Enhanced Linux (SELinux), focusing on its mandatory access control policies and how to troubleshoot policy violations.
- **Cryptography concepts**: We introduced fundamental cryptographic principles, including encryption, decryption, and hashing, which are vital for securing data.

In the rapidly evolving landscape of technology, DevOps and DevSecOps professionals must prioritize security to safeguard systems and data. This chapter equipped you with the necessary skills for protecting sensitive information from unauthorized access and ensuring data integrity and confidentiality. You now are also able to mitigate risks associated with user permissions and to enforce strict access controls.

Collectively, these skills empower professionals to build resilient infrastructures that can withstand potential threats, fostering a culture of security within development processes. The integration of these security practices not only protects assets but also promotes compliance with industry standards, ultimately leading to more robust and trustworthy systems. As cyber threats continue to evolve, the importance of these skills will only increase, making them indispensable for any professional in the field.

CHAPTER 22

Securing Data and Validating Identity Through GPG

GPG (GNU Privacy Guard) skills are essential for DevSecOps professionals, since they enable secure communication and ensure data integrity through encryption and digital signing. Mastering GPG helps manage sensitive information, verifying that code and configurations are authentic and untampered, which is critical for security throughout the development lifecycle.

While multiple GPG implementations exist, this chapter focuses on GNU PG (commonly referred to as GPG), the most widely used in the Linux ecosystem. We'll explore it through hands-on example labs with runnable commands to demonstrate concepts in action.

To avoid distractions from environment setup in each example, we'll establish a baseline lab now.

GPG involves exchanging encrypted and digitally signed information between parties, so we need a personal user to represent you. If you don't have one, create it now.

In this example, we'll create a user named `mcarcano`:

```
sudo adduser mcarcano
sudo passwd mcarcano
```

Note From this point forward, we'll refer to this as **your personal user**.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_22

Next, assign the `practitioner` user to the `devops` group we previously created:

```
sudo usermod -a -G devops practitioner
```

It is also necessary to add your personal user to the `devops` group:

```
sudo usermod -a -G devops mcarcano
```

Note If your personal user already existed and you're logged in with it, since the `usermod` command affects only new sessions, you must disconnect the current session (type `exit` or press CTRL+D) and log in again.

To demonstrate GPG for digitally signing RPM packages, we need a dedicated service user for running builds.

Create the `build` user and add it to the `build` group we created in Chapter 17:

```
sudo adduser build -g build
```

We also need to create the `/opt/build` directory with appropriate permissions:

```
sudo mkdir -m 775 /opt/build
sudo chown build: /opt/build
```

Since we are going to use the `build` user for software build and CI/CD purposes, it is also necessary to configure `git`.

Switch to the `build` user:

```
sudo su - build
```

and configure its git Author identity as follows:

```
git config --global user.email "build@carcano.corp"
git config --global user.name "Carcano SA - BUILD USER"
```

Then exit this sudoed shell:

```
exit
```

To demonstrate backing up and restoring GPG keys, we need to simulate a USB stick mounted at `/media/usb-stick`. For this purpose, just create this pretend mount point as follows:

```
sudo mkdir -m 777 /media/usb-stick
```

This completes the baseline setup we will use as the starting point for all examples in this chapter.

What Is GNU PG (GPG)

GNU PG (GnuPG) is a freely available command-line encryption suite based on the OpenPGP standard and is arguably the most widely used OpenPGP implementation in the world.

PGP, GNU PG, and OpenPGP are related terms that are often confused:

- **PGP** (Pretty Good Privacy) is the encryption software developed by Phil Zimmermann in 1991 that uses a combination of hashing, data compression, symmetric key cryptography, and public key cryptography. It is now a proprietary product owned by Broadcom (formerly Symantec).
- **OpenPGP** is the open standard defined originally in RFC 2440 (1998) and updated in RFC 4880 (2007). In 2024, it was modernized by RFC 9580 (often called "crypto-refresh"), which is now the current standard.
- **GNU PG** is the free, open-source implementation of the OpenPGP standard. It was started by Werner Koch in 1997, with the first stable version released in 1999.

GPG relies on GPG keys: these are security items containing the user ID (i.e., a name and an email address pair), along with an asymmetric key pair. GPG keys can be used to encrypt data or to sign contents – this of course includes signing someone else's GPG key.

Note This chapter contains example GPG statements which should be run from a session not started using `sudo`. If you must use GPG with `sudo`, you must include the `--pinentry-mode loopback` option to ensure password prompts function correctly. The examples assume an offline USB stick is mounted at `/mnt/usb-stick` for secure key storage.

The Web of Trust

Signing someone else's key is a critical part of the GPG protocol: the aim is to securely identify known individuals and assign their keys a level of trust.

As the number of signed keys and assigned trust levels grows, it forms a web of trust (WoT) – a decentralized ensemble of trust relationships.

GPG uses this web to infer the validity of keys automatically, enabling us to automatically trust even GPG keys of people we do not directly know thanks to the trust level inherited from being signed by someone we know and trust.

While technically it is possible to operate GPG even with just a single GPG key, this is not recommended if you want to have that key be part of the web of trust: if that key were compromised, it would affect the web of trust, compromising part of the trust levels. In addition to weakening the `web of trust`, this would have a very bad impact on your own rating within the web of trust.

For this reason, the best practice with GPG is to distinguish between primary keys and subkeys.

Historically, keys were published on a global network of key servers. However, the traditional SKS key server network became largely unreliable following massive "poisoning" attacks in 2019 and legal challenges regarding GDPR compliance. The current industry standard is `https://keys.openpgp.org`, which provides a searchable web interface and better privacy controls

Note To prevent spam, `https://keys.openpgp.org` does not distribute third-party signatures, meaning it facilitates key discovery but does not directly host the web of trust.

GPG Basic Concepts

Before seeing GPG in action, it is necessary to acquire a few basic concepts and familiarize yourself with its security objects.

Primary Keys

GPG primary keys, also known as `master` keys, are self-signed keys and are the most sensitive ones since they are supposed to be used for

- Signing someone else's primary keys
- Generating and signing our own subkeys, which are the keys intended for daily usage

The best practice is to securely store the primary keys offline and securely shred them by overwriting their contents multiple times with random data before removing them from the system.

Primary keys must be restored on the system only when strictly necessary, so only when signing someone else's primary key, or generating secondary keys, or extending the lifetime of the existing ones.

Right after that the primary keys must be securely shredded and removed from the system again.

Note While the OpenPGP protocol technically allows for certification-capable subkeys, current GPG implementations do not support creating them. The rationale is that besides the risk related to using an always available key as the signer, the other parties' signed primary keys' trustworthiness may be questioned if the secondary key is not widely recognized or trusted by others. This can lead to warnings about the signature not being certified. When dealing with establishing trusts, the best practice is to use only primary keys for this.

To see GPG primary keys in action, we can now run a small lab.

Log in to the system as your personal user – in my scenario I'm logging in as `mcarcano`.

Note Don't use `sudo` to switch users: it is mandatory to operate the next statements after performing a real login. It is not necessary a real console login: just an SSH session is fine (i.e., `ssh -l mcarcano localhost`).

The commonly used and easiest way for generating a GPG primary key is using the interactive wizard.

Just for reference only – don't run it now – the command to run for invoking the wizard is

```
gpg --expert --full-gen-key
```

Note If you accidentally run the command, terminate it by pressing `CTRL+C`. The interactive process is straightforward: you just need to follow the wizard, choosing the desired key kind, capabilities, and size, the expiration date, the UID data (real name, email), and the comment describing it. Since the interactive process is simple but less flexible for automation, this book focuses on the more complex yet powerful batch mode, which allows for scripted and reproducible key generation.

Sometimes, especially when embedding statements in batches, it is instead necessary to operate in batch mode, which means running everything within a single statement.

The batch mode is invoked by using the `--quick-gen-key` option as follows:

```
gpg --quick-gen-key \
'Marco Antonio Carcano (Marco Antonio Carcano Primary Key) <marco.carcano@
carcano.corp>' nistp256 sign,cert 10y
```

In the above example I am creating my own ten-year lifetime GPG primary key using ECC with `nistp256` elliptic curve.

The important bit of the output is

```
pub   nistp256 2025-02-26 [SC] [expires: 2035-02-24]
```

4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681

Since it reports `4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681` as the fingerprint of the key, we will use this value in several GPG statements to uniquely identify this key.

Note The output contains also a line starting with `revocation certificate stored as`, which is used to provide the path of the certificate revocation file. That certificate is a critical security object you will use if your certificate would ever be compromised. For this reason you must locate it and save that file offline in a safe offline storage. As an exercise, copy it into the `/mnt/usb-stick` to simulate an offline backup.

Subordinate Keys

GPG subordinate keys, also known as secondary keys, are purpose-specific keys signed by a primary key.

They can be designated for different functions:

- **Encryption (E)**: Used for encrypting messages
- **Signing (S)**: Used for signing documents or messages
- **Authentication (A)**: Used for authentication purposes, such as logging into systems

Subordinate keys are intended to be used for daily operations.

The commonly used and easiest way for generating a GPG Subordinate key is using the interactive wizard.

Just for reference only – don't run it now – the command to invoke the wizard is

```
gpg --expert --edit-key marco.carcano@carcano.corp addkey
```

Note If you accidentally run the command, terminate it by pressing `CTRL+C`. The process is straightforward: you just need to follow the wizard, choosing the desired key kind, capabilities, size, and the expiration date. When the wizard completes, type `save` to permanently store the generated key. Since the interactive process is simple but less flexible for automation, this book focuses on the more complex yet powerful batch mode, which allows for scripted and reproducible key generation.

The batch mode is invoked by using the `--quick-add-key` option as follows:

```
gpg --batch --quick-add-key \
4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681 nistp256 encr 1y
```

In the above example, I am using my GPG primary key with fingerprint `4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681` to sign a new ECC with `nistp256` elliptic curve subordinate key valid for one year.

Note As a required exercise, connect as the `practitioner` user via SSH – for example, using `ssh -l practitioner localhost` – and log in with the password you set during the baseline setup in the previous chapter. Once connected, create an `ECC nistp256` primary key followed by a subordinate key for signing purposes, specifying `rsa` type with `sign` capability. When creating the key, as an email, you can use anything coherent with your environment, such as `practitioner@carcano.corp` in this scenario. After completing the key generation, log out of the `practitioner` session and reconnect via SSH as your personal user – in this example, reconnect as `mcarcano`.

Operating with GPG

We are now ready to have a look at how to operate using GPG.

Note The next statements are supposed to be run after logging in as user `mcarcano`.

Listing Keys

GPG has two distinct options for listing keys.

The `--list-keys` option lists the public keys:

```
gpg --list-keys
```

On my system, the output is

```
/home/mcarcano/.gnupg/pubring.kbx
---------------------------------
pub   nistp256 2025-02-26 [SC] [expires: 2035-02-24]
      4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681
uid           [ultimate] Marco Antonio Carcano (Marco Antonio Carcano
Primary Key) <marco.carcano@carcano.corp>
sub   nistp256 2025-02-26 [E] [expires: 2026-02-26]
```

If instead we want to list the private keys, we must use the `--list-secret-keys` option:

```
gpg --list-secret-keys
```

On my system, the output is

```
/home/mcarcano/.gnupg/pubring.kbx
---------------------------------
sec   nistp256 2025-02-26 [SC] [expires: 2035-02-24]
      4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681
uid           [ultimate] Marco Antonio Carcano (Marco Antonio Carcano
Primary Key) <marco.carcano@carcano.corp>
ssb   nistp256 2025-02-26 [E] [expires: 2026-02-26]
```

Exporting and Importing Keys

Another critical operation to learn is importing and exporting keys.

Public Keys

When dealing with public keys, it's quite common to import someone else's public key to encrypt content that only that user can read or to verify GPG-signed messages.

Similarly, it is very common to export your own public key and share it with others.

As an example, let's see how to export the public key of the currently logged-in user – here, you are supposed to be logged in as your personal user (which is `mcarcano` in my scenario).

The following command shows how to export the public key into the file `/media/usb-stick/mcarcano-gpg.pub`:

```
gpg --export --armor marco.carcano@carcano.corp \
> /media/usb-stick/mcarcano-gpg.pub
```

Note The `--armor` option is used to export an armored key, which is a text-based format: this is the handiest format since it can be easily copied and pasted into any other text document, such as emails.

As an example lab, we are now going to simulate GPG interactions between your personal user and the `practitioner` user.

Let's start by logging in as the `practitioner` user and, using a command like the previous one, export its public key into the file `/media/usb-stick/practitioner-gpg.pub`.

Once done, we need to import the `marco.carcano@carcano.corp` public key from the `/media/usb-stick/mcarcano-gpg.pub` file – still logged as `practitioner`, just run

```
gpg --import /media/usb-stick/mcarcano-gpg.pub
```

Although the key is imported, it is still not trusted. Since that key is obviously not trusted by anybody we trust (we are in a lab and we generated it), it is not part of the web of trust. This means we must assign a trust level to it ourselves. Assigning a trust level to the key is important, since it enables GPG to highlight the trust level each time we run commands involving that key.

Since in this scenario we are pretending to be sure that the key we have imported belongs to the `practitioner` user and that we personally know the user, we can trust the key by running

```
gpg --edit-key '<marco.carcano@carcano.corp>' trust
```

In our simulated scenario, we can set it as `I trust ultimately`, but in real life this really depends on how deeply you trust that key and its owner. You may find more on this topic in the official GPG documentation.

To complete the example scenario, log out and then log in as your personal user (`mcarcano` in my case). Once connected, import the `practitioner` user's public key from `/media/usb-stick/practitioner-gpg.pub`. Since we personally know the practitioner user, trust its key as we did previously.

Note Signature verification and content encryption are really effective only when using trusted public keys.

Private Key

Private keys must be secured by exporting them as soon as they get created, safely storing the backup offline so as to be able to restore them as necessary.

As an example let's see how to export every private key belonging to your personal user.

In my scenario, I'm logging in as the `mcarcano` user and running

```
gpg --export-secret-keys --armor --output /media/usb-stick/marco.carcano@
carcano.corp.sec.asc
```

Note The above statement exports every secret key of the currently logged-in user, not just the primary ones.

As we said, the best practice after exporting the secret keys is to securely shred the primary key's private key and remove it from the system.

Since the private keys are stored in files having the keygrip as the file name, to identify the file to shred and remove, we must first get the primary key's keygrip by running

```
gpg --with-keygrip --list-secret-keys \
4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681
```

Of course, replace `4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681` with your primary key's fingerprint.

Look for the `Keygrip` line with the word `sec` at the beginning of the line – in my case it is

```
Keygrip = B05D736DFC5739EF0478FB02046D909C3709C079
```

Since private keys are stored beneath the `~/.gnupg/private-keys-v1.d` directory, navigate to it as follows:

```
cd ~/.gnupg/private-keys-v1.d
```

We can now securely shred the file by running

```
shred -vz -n 7 B05D736DFC5739EF0478FB02046D909C3709C079.key
```

The above command overwrites the file seven times with random data. When it completes, we can safely remove it as follows:

```
rm -f B05D736DFC5739EF0478FB02046D909C3709C079.key
```

If you list again the private keys, this time the # symbol is shown right after the `sec` word of the line of the secret key we just deleted: this convention is used to inform us that the key exists in the GPG configuration, but it's actually missing on the disk.

When we need to import it again, we will just use the `--import` option.

To complete the lab, let's import the keys we just saved as follows:

```
gpg --import \
/media/usb-stick/marco.carcano@carcano.corp.sec.asc
```

The best practice is obviously to make sure the import was successful – just list again the keys by running

```
gpg --with-keygrip --list-secret-keys \
4EF18252DC13DB5A434FB03E69BA9CE0ECE8B681
```

This time the # symbol meaning the missing private key must have disappeared.

Encrypting And Decrypting Contents

If you work in a confidentiality-focused environment, such as banking or financial fields, it is very common to have to deal with sharing encrypted documents with partners or customers. GPG provides a very easy and convenient way for implementing that.

To see an example of this in action, suppose that since you have no direct access to the `/usr/share/doc/gnupg2/README` file, the `practitioner` user needs to securely share it via NFS, being sure that only you (the `mcarcano` user in this scenario) can access its very sensitive contents.

The `practitioner` user can just encrypt the `/usr/share/doc/gnupg2/README` using your personal user's public key, saving the encrypted document as `README.asc` in the shared directory `/srv/nfs/exports/devops`.

To simulate this, after logging in as the `practitioner` user, the command to run is

```
gpg -e --armor -r marco.carcano@carcano.corp \
--out /srv/nfs/exports/devops/README.asc \
/usr/share/doc/gnupg2/README
```

Note If in real life you need to share the document with multiple recipients, just use the `-r` option followed by the recipient's ID multiple times.

There is no confidentiality problem with the `/srv/nfs/exports/devops` directory being an NFS share, and so its contents can be accessed by any user belonging to the `devops` team: since the `practitioner` user encrypted the `~/README.asc` file using your personal user's public key, then you are the only one able to decrypt it.

Note Depending on your security concerns and operational risks, be wary that, to ease the decryption procedure, by default clear-text metadata about each recipient's key-ID (`marco.carcano@carcano.corp` in this example) is stored within the encrypted file. Although this is handy for automatically selecting the correct key while decrypting, it is also precious information that malicious and evil guys can use to infer who owns that key and maybe use "hard" means to extort it and decrypt the contents. You can prevent the recipient's key-ID from being included by using the `-R` option instead of `-r`.

Let's complete the example scenario by decrypting the document using your personal user's private key.

Log out the `practitioner` user and log in as your personal user (`mcarcano` in this scenario), then decrypt the file as follows:

```
gpg --out ~/README -d /srv/nfs/exports/devops/README.asc
```

This generates the `~/README` file with the unencrypted original contents – you can check it by running

```
cat ~/README
```

Signing Contents

The other GPG application's main purpose is signing contents: this is not limited to signing documents, but also applications.

When dealing with documents, there are three different usage scenarios:

- Generate a GPG-signed document.
- Generate a text GPG signature at the bottom of a document.
- Generate a detached signature file.

Generating a detached signature file can of course be used also to sign applications, as we will see in the next section.

GPG-Signed Document

This method involves creating a signed version of the document in a GPG native format embedding the signature within the document itself.

As an example scenario, let's this time pretend the `/usr/share/doc/gnupg2/README` document does not contain confidential data, but it still contains business-critical information. Since this information is critical, we must be sure to easily detect if anybody tampers with the file, for example, to twist us.

For this reason, the `practitioner` user, instead of just sharing with us a copy of the `README` file, generates a GPG-signed version of it.

To simulate this, after logging in as the `practitioner` user, just run

```
gpg --out /srv/nfs/exports/devops/README.gpg \
--sign /usr/share/doc/gnupg2/README
```

The generated GPG-signed document is saved in the `/srv/nfs/exports/devops` directory as `README.gpg` file.

Since this is a GPG-signed document, we can just verify the GPG signature – if it succeeds, then it means none of our evil competitors have altered it.

Log in as your personal user (`mcarcano` in this scenario) and then just type

```
gpg --verify /srv/nfs/exports/devops/README.gpg
```

The outcome must look as follows:

```
gpg: Good signature from "Practitioner (Practitioner Primary Key)
<practitioner@carcano.corp>" [ultimate]
```

Once the signature is verified, since the document is in GPG format, we can extract the original contents from the GPG-signed file by running

```
gpg --output ~/README.txt --decrypt /srv/nfs/exports/devops/README.gpg
```

In the above example, the original document gets extracted as `~/README`.

Clear-Text Signature

This use case applies only to text documents, adding the signature to the end of the text document. A typical use case is the GPG signature added at the bottom of a mail message.

To see this in action, we can reuse the previous lab scenario, this time generating a clear-text signature at the bottom of the document.

In this case, the `practitioner` user would use the `--clearsign` option in place of the `--sign` one.

To simulate it, we must log in as the `practitioner` user and run

```
gpg --out /srv/nfs/exports/devops/README.asc --clearsign \
 /usr/share/doc/gnupg2/README
```

This time GPG preserves the original text format of the document.

If you have a look at the generated document

```
cat /srv/nfs/exports/devops/README.asc
```

you would see that GPG added at the end of the file

- A clear-text header ("`-----BEGIN PGP SIGNED MESSAGE-----`") that claims that the document is PGP signed
- A clear-text footer ("`-----BEGIN PGP SIGNATURE-----`", "`-----END PGP SIGNATURE-----`") with the actual signature

The statements your personal user (`mcarcano` in this scenario) would run to verify the signature and extract a copy of the original document are the same as in the previous example – of course remember to run it after logging as your personal user.

Detached Signatures

A detached signature is a separate file that contains the signature for a specific document, leaving the original document unchanged. A typical use case provides a GPG detached signature along with software packages, so as to let the user verify that it is not tampered with.

To see this concept in action, we can verify the GPG detached signature of a publicly available software such as Gitea.

First, download the Gitea package as follows:

```
wget \
https://dl.gitea.com/gitea/1.23.1/gitea-1.23.1-linux-amd64
```

Then, download its detached signature file:

```
wget \
https://dl.gitea.com/gitea/1.23.1/gitea-1.23.1-linux-amd64.asc
```

To verify the signature, we need to have the GPG public key used for signing the package in our GPG keyring.

Sometimes the vendor ships this GPG public key along with the application or publishes it on their corporate website. Since Gitea does not follow this practice, we must retrieve its GPG key from the key servers.

In such a scenario, we must first guess the fingerprint of the key that was used for signing:

```
gpg --list-packets < gitea-1.23.1-linux-amd64.asc
```

The important bit from the output is this:

```
hashed subpkt 33 len 21 (issuer fpr v4 CC64B1DB67ABBEECAB24B6455FC346329753F4B0)
```

Which reveals that the GPG key's fingerprint is `CC64B1DB67ABBEECAB24B6455FC346329753F4B0`.

We can now retrieve the public certificate directly from the online GPG key servers. Just run

```
gpg --keyserver keys.openpgp.org --recv
CC64B1DB67ABBEECAB24B6455FC346329753F4B0
```

Note Depending on the key servers' load, this operation may take some time to complete, so don't worry if the statement is not immediately complete.

We are ready to verify if the Gitea application matches its detached signature:

```
gpg --verify gitea-1.23.1-linux-amd64.asc gitea-1.23.1-linux-amd64
```

The important bit from the output is

```
gpg: Good signature from "Teabot <teabot@gitea.io>" [full]
```

If instead we get

```
gpg: BAD signature from "Teabot <teabot@gitea.io>" [full]
```

the signature mismatches their key. In this case, the software has been tampered with.

In this specific context, don't bother instead of the warning:

```
WARNING: This key is not certified with a trusted signature! There is no
indication that the signature belongs to the owner.
```

That is normal and it is expected, since you don't have any key in the web of trust enabling the automatic inference of the trust for their key. In addition to that, most of the time, software is signed directly using primary keys.

Anyway, for the sake of completeness, if you decide to trust the vendor's GPG public key, you must sign it and set it as ultimately trusted.

First, sign it as follows:

```
gpg --edit-key '<teabot@gitea.io>' sign
```

Then, trust it:

```
gpg --edit-key '<teabot@gitea.io>' trust
```

Signing Applications

As mentioned, it is possible to use GPG also to sign applications.

There are only two possible scenarios:

- Detached GPG signature
- Package-specific embedded GPG signature – for example, GPG-signed RPM or GPG-signed DEB packages

GPG Detached Application's Signatures

Same way as for generic documents, it is possible to generate a detached signature for an application binary or for the package providing an application (e.g., a zipped tarball). The signature file can then be published on the repository, along with the application or application package the signature file was generated from.

To demonstrate this and complete the use case described in the previous scenario where we checked the signature of the Gitea application, we can simulate the process run by the `practitioner` user to generate the GPG detached signature for the fictional `myfancysoftware` application.

To set up this lab, first log in as the `practitioner` user and copy the `echo` command-line application pretending it is the fictional `myfancysoftware` software they developed:

```
cp /usr/bin/echo /srv/nfs/exports/devops/myfancysoftware
```

You can now run the lab and generate the detached GPG signature file of the `myfancysoftware` application – just run

```
gpg --armor --output \
```

```
/srv/nfs/exports/devops/myfancysoftware.asc \
--detach-sig /srv/nfs/exports/devops/myfancysoftware
```

To complete the example scenario, let's now assume that the `practitioner` user is sharing the `myfancysoftware` application along with the GPG signature file: any other user can now verify the application's file integrity by checking the signature file.

To see this in action, log in as your personal user – so `mcarcano` in this example.

Once done, just run

```
gpg --verify /srv/nfs/exports/devops/myfancysoftware.asc /srv/nfs/exports/
devops/myfancysoftware
```

Package-Specific Embedded GPG Signature

As anticipated, some package management utilities, such as RPM and DEB, support embedding GPG signatures directly into the packages themselves.

To show this process in action, we illustrate how to GPG sign RPM.

GPG Signing RPM

In Chapter 17, we said that the best practice when creating RPM packages is to digitally sign them using GPG. We now have all the necessary skills to do this.

Note Whether you are providing software or just using third-party applications, using only GPG-signed RPMs is definitely a security best practice: if you are the customer, you are sure what you are installing is not tampered with. If you are the software provider, you are showing your customers that you are meeting security best practices to protect them from installing tampered software.

As anticipated, the Fancymonitor RPM packages we created in the previous chapter are not GPG signed. We can easily check this by verifying their signatures: switch to the user we used when building by running `make rpm` in Chapter 17 – this is the `vagrant` user

if you are using a VM deployed with `Vagrant` or the user you generally used to run your labs. Once done, just run

```
rpm -vK ~/rpmbuild/{RPMS/noarch,SRPMS}/*.rpm
```

The output for every RPM package should look as follows:

```
Header SHA256 digest: OK
Header SHA1 digest: OK
Payload SHA256 digest: OK
MD5 digest: OK
```

As you can see, the only available signatures are digests: this proves that the packages (as expected) are not GPG signed.

The RPM Signing Process

Since the signing feature for the `rpm` command-line utility is provided by the `rpm-sign` RPM package, we must first install it as follows:

```
sudo dnf install -y rpm-sign
```

Adding the GPG signature to the RPM packages is the last step immediately before publishing the packages into a software repository. It is usually part of a multistep automated pipeline typically run by an automation using a dedicated system user.

To illustrate the GPG signing process, let's suppose we have a fictional pipeline with an RPM GPG signing step.

To mock this up, we must set up the lab in the same state it would be when this pipeline reaches the GPG signing stage.

Since we already have the `build` user from a previous lab, we design our fictional pipeline to run as this user, executing the process in the `/opt/build` directory.

So, first things first, we must move the Fancymonitor project directory to the `/opt/build` directory.

```
mv ~/devsecops-pillars/fancymonitor /opt/build
```

To match the user running the pipeline, we must reassign the ownership to the `build` user:

```
sudo chown -R build: /opt/build
```

We are now ready to start illustrating the GPG signing process.

First, switch to the `build` user:

```
sudo su - build
```

Then change to the `/opt/build/fancymonitor` directory as follows:

```
cd /opt/build/fancymonitor
```

The changes implementing the GPG signing process are implemented in the `f-gpg-sign` branch – check it out as follows:

```
git checkout f-gpg-sign
```

To see these changes, let's compare this branch with its parent:

```
git diff f-rpm
```

Spend a little time checking them: as you can see, it just defines the `sign` target in the `Makefile`.

Despite the `Makefile` containing every necessary statement for signing the RPM, anyway our lab setup is still not complete, since the `build` user is still missing the `<build@carcano.corp>` GPG key.

For this purpose, let's generate an RSA 2048-bit primary key:

```
gpg --pinentry-mode loopback --quick-gen-key \
'Build (Build Primary Key) <build@carcano.corp>' \
rsa2048 sign,cert 10y
```

We must now export the public key as follows:

```
gpg --armor --export > ~/CARCANO-BUILD-key
```

Since this user has just one GPG key, in the export statement it was not necessary to supply the key's ID.

Note In a real-world scenario, the `CARCANO-BUILD-key` must then be published in the same repository providing the RPM packages so that the users

downloading the software can easily download the GPG public key and use it to verify the signature.

The GPG environment setup is complete now, so we can proceed with signing the RPM.

First, build the RPM package as follows:

```
export RELEASE=0.1.0
make rpm
```

Then, add the GPG signature by running

```
make sign
```

when prompted for the GPG key's unlocking password, type it.

Using GPG Agent to Automatically Unlock Private Keys

As you are certainly thinking, in the real world, it is impractical to use an interactive password in a fully automated pipeline.

The way out is to register the unlocking password in the `gpg-agent` process.

This is a component that intercepts the GPG's unlocking request and asks interactively for the password, caching it in memory, avoiding requesting it again on the next usage.

It is possible to configure it to preset the password: since this feature is disabled by default, we must enable it by adding the following directive to the `~/.gnupg/gpg-agent.conf` file:

```
allow-preset-passphrase
```

We must then reload the `gpg-agent` setting – just kill and relaunch it as follows:

```
gpgconf --kill gpg-agent
gpg-connect-agent reloadagent /bye
```

We are almost ready to pre-set the GPG key's unlocking password – we just need to get that key's Keygrip:

```
gpg --with-keygrip --list-secret-keys  | grep Keygrip
```

Now, to simulate a batch environment, let's set the password of the GPG key into the GPG_PASSPHRASE variable – in this specific case, we are using the read statement to avoid the password being stored in the shell's history:

```
read -s GPG_PASSPHRASE
```

We can now register the password into the gpg-agent as follows – mind to use the actual keygrip of your key:

```
/usr/libexec/gpg-preset-passphrase --preset --passphrase \
"${GPG_PASSPHRASE}" 72AF40891E8FA3A0676F4D813A3663C7CB9101AA
unset GPG_PASSPHRASE
```

Note Please note how we are unsetting the GPG_PASSPHRASE variable right after having registered it to mitigate the risk of security breaches.

The environment is now set up to run everything in batch mode.

We can finally have a go with this non-interactive setup. First, let's clean up and regenerate the RPM packages:

```
make clean rpm
```

Then, let's add the GPG signature:

```
make sign
```

This time the process must be completed without asking us for the GPG password to unlock the signing key.

The RPM packages are now suitable to be published in the software distribution repository, enabling the users to download and install them only after having verified the GPG signature.

Note RPM uses its own GPG key ring. In an automated pipeline, it is anyway a best practice to install the GPG key in the RPM's GPG keyring and run the rpm statement to verify that not only the package is signed, but also that the signature matches.

Work with GPG-Signed RPM

As you certainly inferred, GPG is a critical part of the security features for RPM packages. This means that you are supposed to know the operational processes to securely use them.

Check if an RPM Package Is GPG Signed

The very first operation to learn is how to ensure an RPM package is GPG signed.

As anticipated, this requires running the `rpm` command with the `-K` option. In our previous run, the output was missing the GPG signature. Since in the meantime we signed the Fancymonitor's RPM packages, we can run again the same statement and see the changed output:

```
rpm -vK ~/rpmbuild/{RPMS/noarch,SRPMS}/*.rpm
```

As expected, this time we have a new entry in the first line:

```
Header V4 RSA/SHA256 Signature, key ID 04f9cad6: NOKEY
```

As you can see, this line is about a key signature, confirming the RPM has been GPG signed using the key with the ID `04f9cad6`.

Untrusted Signatures

The GPG signature line from the above output ends with the `NOKEY` word: you must be very careful when you get this, since the `NOKEY` word is used to warn us that the key is not listed in the RPM keyring, making it impossible to check if the GPG signature is valid.

In our scenario, this is expected, since signing RPM packages does not require having the GPG key installed in the RPM's GPG keyring.

Import a GPG Key in the RPM GPG Keyring

To be able to verify the GPG signature of an RPM package, it is necessary to import the GPG public key used to sign it into the RPM GPG keyring.

Continuing with our last example, we must first import the Carcano build user's GPG key in the RPM keyring.

This operation requires administrative privileges, so we must first switch to a user granted to run administrative commands using `sudo`.

Once done, run

```
sudo rpm --import /home/build/CARCANO-BUILD-key
```

Valid Signatures

We can now run the `rpm` statement to verify the GPG signature – this time, it must not emit the `NOKEY` warning.

For example, to verify this, switch back to the `build` user, then run

```
rpm -vK ~/rpmbuild/{RPMS/noarch,SRPMS}/*.rpm
```

This time the output must be as follows:

```
Header V4 RSA/SHA256 Signature, key ID 04f9cad6: OK
```

If the RPM GPG signature is valid, then it is safe to proceed with installing the RPM packages being confident the RPM packages were not tampered with.

Operating the RPM's GPG Keyring

As we said, the `rpm` command-line tool manages its own GPG keyring containing the trusted public GPG keys used to verify the RPM package's signature.

These keys can be listed by running

```
rpm -q --queryformat "%{SUMMARY}\n" gpg-pubkey
```

For example, on my Oracle Linux system, the output is

```
Oracle Linux (release key 1) <secalert_us@oracle.com> public key
Oracle Linux (backup key 1) <secalert_us@oracle.com> public key
Build (Build Primary Key) <build@carcano.corp> public key
```

As you can see, in addition to the first two keys, which are the ones provided by the Oracle Linux distribution, we also have the `<build@carcano.corp>` key we just imported.

GPG Check with YUM/DNF

YUM (Yellowdog Updater, Modified) and its successor DNF (Dandified YUM) are package management tools used in RPM-based Linux distributions. They serve as front-end tools for managing software packages in the RPM (Red Hat Package Manager) format.

Yum and DNF simplify the process of installing, updating, and removing software packages automatically handling dependencies, ensuring that all required packages are installed or updated alongside the requested software.

Software is fetched from the YUM or DNF repositories configured in the system. To implement best practices, they can (actually should) be set up to enforce RPM GPG key checking on a per-repository basis.

Note Since the concepts in this paragraph apply to both YUM and DNF, to avoid continuously repeating both names, by this point onward we just use the word `yum` to refer to both.

The `/etc/yum.repos.d` directory contains the `yum` repository-specific configuration files: these are used to define each single repository along with its specific settings, such as the repository URL.

If we look at any of them, among the various settings, there are also two GPG-related ones:

```
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-oracle
```

The `gpgcheck` parameter is used to enforce the GPG check of the RPM packages: if it is set to `1`, then only GPG-signed RPM packages with a valid signature can be installed from that repository.

The `gpgkey` directive is instead used to specify the path to the GPG key used to sign every package provided by the repository, so as to be able to automatically install it on the system if it is missing. The GPG keys imported in the RPM GPG keyring are typically stored beneath the `/etc/pki/rpm-gpg` directory.

If GPG check is enabled, when the RPM packages are downloaded, `yum` ensures they are signed, guesses the GPG key used to sign them, and checks if it is already in

its keyring. If it isn't, it imports the key from the path defined by the gpgkey setting, interactively asking to confirm you want to trust it. If any of these steps were not successful, yum immediately aborts the process.

After this, it performs the rpm installation: if gpgcheck is set to 1, it requests the rpm install process to also perform the GPG check. If the signature check fails, then the transaction is immediately aborted.

As a real-life example, let's suppose we want to add the PostgreSQL's YUM repository. Since they provide a GPG-signed RPM, we first need to download from https://apt.postgresql.org/pub/repos/yum/keys the GPG key suitable for our distribution.

In this example we are downloading the RPM GPG key for Red Hat family aarch64:

```
sudo wget -P /etc/pki/rpm-gpg \
https://apt.postgresql.org/pub/repos/yum/keys/PGDG-RPM-GPG-KEY-AARCH64-RHEL
```

Then, let's create the yum repository configuration file for the PostgreSQL version we want to use.

The best practice is to configure it using a file dedicated to the specific product. For example, for PostgreSQL 17, create the /etc/yum.repos.d/pg17.repo file with the following contents:

```
[pgsql17]
name=PostgreSQL 17 ($basearch)
baseurl=https://apt.postgresql.org/pub/repos/yum/17/redhat/rhel-9-
$basearch/
gpgkey=file:///etc/pki/rpm-gpg/PGDG-RPM-GPG-KEY-AARCH64-RHEL
gpgcheck=1
enabled=1
```

Note In modern yum versions it is possible to also set a remote URL as gpgkey: this means you can avoid the previous wget statement and just set https://apt.postgresql.org/pub/repos/yum/keys/PGDG-RPM-GPG-KEY-AARCH64-RHEL as gpgkey.

We can now run yum statements to install the desired RPM packages as usual. For example:

```
sudo yum install postgresql17-libs
```

Yum automatically loads the `pgsql17` repository's GPG key and asks if we want to import it into the RPM's GPG keyring: just answer `yes` and go on.

Best Practices for GPG Key Management

Before closing this chapter, it is certainly worth the effort to mention the best practices for operating with GPG:

- Do not use primary keys for daily operations: use subkeys.
- Store the encrypted primary key in offline media, such as a USB drive, and store it off-site in a secure storage.
- Perform regular backups of your encrypted keys, storing them in offline media, such as a USB drive, and store it off-site in a secure storage.
- Assign expiration dates to subkeys to limit their validity – this helps to mitigate risks if a subkey is compromised.
- Create and securely store a revocation certificate for each primary key: this can be used to revoke any key if it is lost or compromised.
- Periodically rotate your subkeys to enhance security: this involves generating new subkeys and revoking old ones.
- When signing other users' keys, pay attention when checking if the key you are signing really belongs to the claimed entity, and sign only keys from people you directly know.
- Protect your keys always using strong passphrases.

Summary

In this chapter, we learned how to operate with GPG for enforcing data confidentiality and data integrity through encryption and signing. More specifically we saw how to manage GPG keys securely, securely backing them up offline and restoring them according to best practices. We also learned the necessary concepts to securely trust the

keys according to the specifications of the web of trust. We experimented not only with how to encrypt and decrypt documents but also how to properly exploit the GPG signing capabilities to sign and verify the signature not only of documents, but also of packages. Finally, we delved into GPG-signed RPM packages, explaining not only how to operate with a GPG-signed RPM but also how to sign RPM packages.

These skills are vital for DevSecOps professionals as they enhance the security of communications, protect sensitive data, and ensure the authenticity and integrity of software, thereby fostering a secure development environment.

CHAPTER 23

X.509 Certificates and PKI

In this chapter, we delve into X.509 certificates and Public Key Infrastructure (PKI): these are essential skills for DevOps and DevSecOps professionals, as they provide the foundation for secure communications and identity verification in digital environments.

Proficiency in PKI allows for the management of X.509 certificates, including their issuance, renewal, and revocation, which is crucial for maintaining trust in a secure ecosystem. These skills also facilitate secure authentication and authorization processes, enabling secure access to resources and services.

In addition to that, X.509 certificates and PKI are the pillars of SSL/TLS, which enable the implementation of secure connections, ensuring data confidentiality and integrity during transmission.

Overall, expertise in X.509 certificates and PKI are vital for establishing a robust security posture, protecting sensitive data, and ensuring compliance with security standards in the development lifecycle.

This chapter contains several example statements you can run to see the explained concepts in action.

To streamline the labs and avoid repetitive setup for each example, we establish a baseline environment now.

Some practical examples require connecting to services using their Fully Qualified Domain Names (FQDNs). Since these examples run on the same machine as the services, we use the `/etc/hosts` file for hostname resolution to avoid making invasive changes to the network configuration.

Add the following line to the /etc/hosts file:

```
127.0.0.1 www.t1.carcano.corp www.t2.carcano.corp dnf.t1.carcano.corp
```

If a line starting with 127.0.0.1 already exists, append the hostnames `www.t1.carcano.corp` `www.t2.carcano.corp` dnf.t1.carcano.corp.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_23

This entry maps the hostnames `www.t1.carcano.corp`, `www.t2.carcano.corp`, and `dnf.t1.carcano.corp` to the IP address `127.0.0.1`, which is the loopback interface (localhost) pre-configured on every system.

Note Before going any further, change to this **chapter's workspace directory** by running `cd ~/devsecops-pillars/ch23`.

X.509 Certificates

X.509 certificates are a standard format for public key certificates, which are used in various security protocols, including SSL/TLS, to secure communications over the Internet.

Leveraging public key cryptography, X.509 is the security-related part of the X.500 standard, which was developed to address several key needs in the realm of digital security.

Along with the Public Key Cryptography Standards (PKCS) and the IETF RFCs for Certificate Revocation Lists (CRLs) and the Online Certificate Status Protocol (OCSP), X.509 is one of the cornerstones of Public Key Infrastructure (PKI).

Self-Signed Certificates

The simplest kind of X.509 certificate is the self-signed certificate: this is a self-issued certificate that is signed by itself.

The issuer and its signature, as in real-life, are critical information used to decide on the trust to grant to the certificate: the actual trust on a certificate derives from the trust we have on who signed it.

Since it is self-signed, this kind of certificate is by default not trusted.

Note Using self-signed certificates is strongly discouraged in production systems, as they hinder the use of a Public Key Infrastructure (PKI) for automatic trust validation of certificates. Instead, they are commonly used in testing scenarios, such as mock environments, labs, or demonstrations.

Generate the Private Key

The first part of the process for generating an X.509 certificate is always generating an RSA or an ECC asymmetric key pair.

As an example, let's generate an unencrypted 2048 bits RSA key:

```
openssl genrsa -out wildcard.t1-t3.carcano.corp.key 2048
```

Let's have a look to the generated key:

```
openssl rsa -noout -text -in wildcard.t1-t3.carcano.corp.key
```

In this example, only because we will use it for requesting a service certificate, we created an unencrypted private key. Anyway, mind that when dealing with other kinds of use cases, such as users' identity certificates, the best practice is to encrypt the generated key on the fly, for example, by adding the `-aes256` option to have it encrypted using the AES 256 cipher.

Note Encrypted keys require a decryption password each time they are accessed: this means that applications using them must be able to decrypt them by using the decryption password. Anyway, mind that there are many applications around which cannot decrypt encrypted keys. In such scenarios, you must forcefully use unencrypted private keys, securing them by restricting read access to only the service accounts used to run the application. However, be wary that this approach has the downside that any user with administrative rights can also access these keys. To mitigate this problem some applications support more secure containers for certificates and keys, such as PKCS#12 or Java KeyStore (JKS), which provide encryption and better management of sensitive keys. Anyway, these too are still cosmetic solutions: unless the decryption password is interactively entered (which is impractical), they require the decryption passphrase to be stored in their configuration file or in a separate password file.

Generate the Certificate Signing Request (CSR)

X.509 certificates are generated from a Certificate Signing Request (CSR), which is a file containing metadata about the entity requesting the certificate specifying claims such as the subject's name, organization, the common name, and so on. In addition to that, the CSR also contains the public key from the asymmetric key pair we just generated.

The claims to be stored in the CSR are put in the so-called subject string: this contains several standard attributes as key/value.

Table 23-1 summarizes the most used ones.

Table 23-1. *X.509 most used subject string keys*

Key	Description
CN	The common name we desire to claim: when dealing with server certificates, it is the server's FQDN. When dealing with client certificates, it can be the machine's FQDN or username either of a human or of a service.
C	The ISO-3166 alpha-2 country code of the organization owning what is claimed in the CN key. For example: CH
O	The name of the organization owning what is claimed in the CN key. For example: Carcano SA
OU	The organizational unit inside the organization which is in charge of what is claimed in the CN key. For example: Digital Services
ST	The state where the organization is established: For example: Tessin
L	The locality where the organization is established. For example: Lugano

As an example use case, let's suppose we need to generate a certificate to be used for labs and demos, valid throughout each of our three different testing environments (`t1`, `t2`, `t3`) and for the `127.0.0.1` IP address.

In such a scenario, we need to generate the CSR for a wildcard certificate: this is a type of certificate that is valid for a domain and all its subdomains at a specific nesting level.

The wildcard token is typically represented with an asterisk (*) in the Common Name (`CN`) field.

Since the CSR generation statement supports only a single Common Name, we cannot specify all these domains and the `127.0.0.1` IP address.

So, just for starting, let's generate the wildcard certificate valid throughout the whole `t1.carcano.corp` domain owned by `Carcano SA` established in Switzerland (`CH`):

```
openssl req -new -key wildcard.t1-t3.carcano.corp.key \
-out wildcard.t1-t3.carcano.corp.csr \
-subj "/C=CH/O=Carcano SA/CN=*.t1.carcano.corp"
```

Note As you can see, the syntax of the `-subj` string parameter requires separating the attributes using the / character.

We can see how the generated CSR looks like by running

```
openssl req -in wildcard.t1-t3.carcano.corp.csr -noout -text
```

Generate the Self-Signed Certificate

As you can see, as expected, the CSR contents does not meet our requirements, since we also wanted to include the whole `t2.carcano.corp` and `t3.carcano.corp` domains, along with the IP address `127.0.0.1`.

Since the CSR is valid only for the whole `t1.carcano.corp` domain, we must still add `t2.carcano.corp` and `t3.carcano.corp` to the final certificate.

We can provide them by exploiting the `v3CA extensions`, which enable us to provide additional claims using the `Subject Alternative Names` key.

When dealing with `openssl`, extensions are added using a `v3CA extensions` file: this file is used not only to provide extensions, but even to specify additional constraints we want to be in the generated certificate.

An example of a v3CA extension file is the `v3.ext` in the chapter's workspace.

It contains the following directives:

```
subjectKeyIdentifier   = hash
authorityKeyIdentifier = keyid:always,issuer:always
basicConstraints       = CA:FALSE
keyUsage               = digitalSignature, nonRepudiation, keyEncipherment,
dataEncipherment, keyAgreement, keyCertSign
subjectAltName         = DNS:*.t1.carcano.corp, DNS:*.t2.carcano.corp,
DNS:*.t3.carcano.corp, IP:127.0.0.1
```

As you can see, the `subjectAltName` parameter contains multiple wildcard domains prefixed by the `DNS:` word as well as the `127.0.0.1` IP address prefixed by the `IP:` word. Please mind how, as required by the standard, although the `t1.carcano.corp` is already mentioned by the `CN` key, we also included it in the `subjectAltName` key.

Note The Subject Alternative Name (SAN) is an X.509v3 extension which allows specifying additional names beyond the Common Name (CN). This enables specifying multiple Fully Qualified Domain Names (FQDNs), wildcard domains, or even IP addresses as necessary.

Once done with the above file, just run the following statement:

```
openssl x509 -req -in wildcard.t1-t3.carcano.corp.csr \
-signkey wildcard.t1-t3.carcano.corp.key \
-out wildcard.t1-t3.carcano.corp.crt \
-sha256 -extfile v3.ext -days 3650
```

The outcome is a self-signed certificate with a SHA256-hashed signature lasting for 10 years (3650 days).

We can have a look at the contents of the generated certificate by running

```
openssl x509 -noout -text -in wildcard.t1-t3.carcano.corp.crt
```

Certificate Validation

The first thing to learn when operating certificates is always their validation: if any of the validation checks fail, then the process is aborted, and the certificate is reported as invalid.

This means that, for example, when dealing with TLS, the processing of the HTTP protocol has not even started.

To illustrate the certificate validation process with the certificate we just generated, we simulate the most classic of the use cases: a TLS session.

Note TLS stands for Tunnel Level Security: it is a protocol leveraging X.509 certificates to establish an encrypted, confidential secure channel between two network peers. We will discuss it in detail in Chapter 25.

To spin up a TLS server, we use the `https.py` from this chapter's workspace directory: it is a Python script that provides an example of how to implement a simple TLS server in Python.

Just launch it as follows:

```
./https.py -c wildcard.t1-t3.carcano.corp.crt \
-k wildcard.t1-t3.carcano.corp.key
```

This script spins up a very basic HTTPS service on port `8443`, enabling clients to download any file in its working directory: in our example, since the `v3.ext` file is in the same directory as the `https.py` we just launched, we can just run a statement for downloading it.

Leave the `https.py` process running and open another terminal before continuing.

Checking If the Certificate Is Trusted

The first of the checks performed is making sure the certificate is trusted: we can see it in action by simply running a `curl` call.

Since, as we said, self-signed certificates are untrusted by design, the following statement is expected to fail (run it from the other terminal we just opened):

```
curl https://www.t1.carcano.corp:8443/v3.ext
```

`curl` indeed fails with the following error:

```
curl: (60) SSL certificate problem: self-signed certificate
```

When dealing with self-signed entity certificates like the one we generated, the only way to have a peer trust it is off-band providing to the peer a copy of the certificate and launching the application we want to run explicitly marking that certificate as trusted.

Let's pretend we got the certificate off-band and that we saved it as `~/devsecops-pillars/ch23/wildcard.t1-t3.carcano.corp.crt`. We can now retry the `curl` statement, this time explicitly trusting the certificate using the `--cacert` option followed by the path to the file containing it:

```
curl --cacert \
~/devsecops-pillars/ch23/wildcard.t1-t3.carcano.corp.crt \
https://www.t1.carcano.corp:8443/v3.ext
```

This time everything worked smoothly as expected.

Note During the certificate validation process, the whole certificate chain is checked. In this case, we are using it to validate a self-signed certificate, so this is a corner case. We will see the complete use case shortly.

Exit this terminal by typing `exit` and go back to the first terminal from where we launched the `https.py` script.

Note Remember to terminate the running `https.py` instance by pressing CTRL+C.

Certificate Lifetime Validation

Once the trust chain is confirmed, the client checks the validity period of the certificate. This involves verifying that the current date and time fall within the `notBefore` and `notAfter` fields of the certificate: if it isn't, then the validation fails.

The previous `curl` call we did obviously passed this check; otherwise we would have got an error message reporting the TLS server certificate was expired.

Mind that, when necessary, we can easily inspect the certificate's lifetime by using the `-dates` parameter as follows:

```
openssl x509 -in wildcard.t1-t3.carcano.corp.crt -noout -dates
```

In our example the output is

```
notBefore=Feb 22 21:31:15 2025 GMT
notAfter=Feb 20 21:31:15 2035 GMT
```

CA-Signed Certificates

As we saw, the shortcoming of self-signed certificates is that we must off-band provide a copy of the certificate to the client so that it can load it and mark it as trusted.

This is a cumbersome shortcoming even in quite small private environments, so it will never fit public Internet use cases.

To overcome these limitations, it is necessary to switch to CA-signed certificates: in this case the certificate is generated by the Certificate Authority (CA), and the validation process happens in the context of a Public Key Infrastructure (PKI).

Certificate Authority (CA)

A Certificate Authority (CA) is a trusted entity that issues digital certificates based on Certificate Signing Requests (CSRs), signing them with its own private key.

CA Certificates

The certificates owned by Certificate Authorities are the only ones suitable for signing other certificates: this right is granted by having the `CA basicConstraint` extension set to `true`.

Based on their tier level, CAs can be divided into

- **ROOT CA**: These CAs own self-signed CA certificates. These certificates are often called ROOT certificates.
- **Intermediate CA**: These CA certificates are signed by other CAs, no matter if the signing certificate is from a ROOT CA or from another intermediate CA. Intermediate CA certificates are often called intermediate certificates.

Public Key Infrastructure (PKI)

Certificates signed by CAs are one of the cornerstones of Public Key Infrastructure (PKI), enabling a trust verification mechanism. A Certificate Authority can sign

- Entity certificates (often called leaf): these are the actual services or client certificates
- Intermediate CA certificates

Certificate Chain Validation

The PKI leverages on the trust derived by the certificate chain validation process: this mechanism builds a chain of certificates starting from the `leaf` certificate back to the `root`, chaining them by matching the issuer in one certificate with the subject in the other CA certificate: if any CA certificate matching is found, then the process is repeated this time focusing on the found CA certificate, until a ROOT CA certificate is found. When it is found, the ROOT CA certificate is checked to be listed among the ones in the `Trust Store`; otherwise, the chain is deemed untrusted.

The Truststore

It is a repository of trusted ROOT certificates used during the certificate chain validation process. As said, to have a certificate chain validation succeed, it is necessary to have its ROOT CA certificate in the truststore.

Publicly available Certificate Authorities' ROOT certificates have their ROOT certificates pre-installed in the system-wide truststore, which is also periodically updated by the system. This ensures that every certificate signed by these global CAs is automatically trusted.

Note Despite the existence of globally recognized public Certificate Authorities (CAs), if needed, you can set up your own private CA and issue certificates signed by it. However, this obviously requires being able to provide to the clients that specific ROOT CA certificate, installing it in their truststore. In addition to that, if you run your own private CA, always remember that the best practice is to never issue a certificate outlasting the lifetime of the certificate used to sign it. As a warning, for the sake of completeness, please note that, by leveraging the mainstream supply chain, it is technically possible to have rogue CAs intentionally inserted in the system's truststore. This may sound appealing, for example, by, let's say, "not so fair" governments, to monitor people's behavior.

Be wary that if you add a ROOT certificate to the system-wide truststore, that certificate becomes automatically trusted from every application in your system: although this is often the desired goal, sometimes it is needed to have selected ROOT certificates trusted only by one or more specific applications. In such a need, it is possible to generate a custom truststore dedicated to that application, providing the path to it through a dedicated command-line option or configuration setting.

Note While it is technically possible to add intermediate CA certificates to a truststore to avoid providing them off-band, this is not considered a best practice and comes with several drawbacks, including increased complexity, operational risks, maintenance overhead, and general security risks.

Generate the Private Key

Note In this lab, we will request a server certificate with CN `www.carcano.ch` to Let's Encrypt, a globally available CA. If you want to try this example yourself, you must of course replace `carcano.ch` with a global top-level domain you own and you control the DNS.

Even when dealing with CA-signed certificates, the first part of the process is still generating an asymmetric key pair: since in the previous example we generated an RSA key pair, this time we are generating an ECC asymmetric key pair.

Note Generating an ECC key pair requires specifying the elliptic curve to use: to know the ones available on the system, we must first run `openssl ecparam --list_curves` statement.

As an example, let's create an `AES256`-encrypted ECC key pair using the `secp384r1` curve as follows:

```
openssl ecparam -name secp384r1 -genkey | \
openssl ec -out www.carcano.ch.key
```

Certificate Signing Request (CSR)

Let's go on with our example and generate a CSR for the www.carcano.ch host:

```
openssl req -new -key www.carcano.ch.key -out \
www.carcano.ch.csr -subj "/CN=www.carcano.ch"
```

It's certainly interesting having a look at the contents of the generated CSR:

```
openssl req -in www.carcano.ch.csr -noout -text
```

Submit the CSR to a Certificate Authority

To obtain a signed certificate, the justly created Certificate Signing Request (CSR) must be submitted to a Certificate Authority (CA) that, after performing the proper identity validation, will issue a signed certificate.

This process can vary depending on the CA. Most of them, whether they are global providers or software applications used to run your own private CA, validate your identity by requiring you to log into a web portal. Once logged on, it is typically possible to upload the CSR, with the option to specify additional details, such as Subject Alternative Names (SANs).

This approach is anyway impractical while requesting certificates using batch processes, so over the years, several protocols have been developed for automatic certificate enrollment. Examples include the Simple Certificate Enrollment Protocol (SCEP), proprietary protocols like those used by Cloudflare's PKI and TLS Toolkit, and the Automated Certificate Management Environment (ACME) protocol. These protocols facilitate the automation of the entire certificate lifecycle, including the renewal of certificates as they approach expiration.

Given its widespread adoption, we will focus on the example of submitting a CSR to Let's Encrypt, a publicly recognized CA that offers free certificates and implements the ACME protocol.

Submit to Let's Encrypt Using the ACME Protocol

Note This lab requires you to use the `host` command-line tool – if it's not available on your system, you must install it by running `sudo dnf install -y bind-utils`.

The Let's Encrypt client component implementing the ACME protocol is called `certbot`. We can install it by running

```
sudo dnf install -y certbot
```

Once installed, we can use it to submit the CSR we previously generated as follows:

```
sudo certbot certonly --manual --csr www.carcano.ch.csr \
--agree-tos --preferred-challenges dns --no-eff-email \
--email devops@carcano.ch
```

where

- `--csr` is the path to the CSR file.
- `--email` is the email to receive status updates about the shipped certificate.
- `--preferred-challenges dns` selects dns as the identity verification process: in this process Let's Encrypt provides a DNS TXT record with a challenge value that must be added to the domain the requested certificate belongs to. Let's Encrypt will then check the existence and value of such a DNS record on the domain's authoritative DNS, issuing the certificate only after a successful check.

Once the above statement is run, `certbot` will provide us with the specific DNS TXT record to be added to the `carcano.ch` authoritative DNS along with the value to set.

In this example, the DNS TXT record to add is `_acme-challenge.www.carcano.ch`: just add it with the value provided by `certbot` and wait a bit for this record to propagate.

Note You can monitor the propagation using the `host` command-line tool by opening another terminal and running: `host -t TXT _acme-challenge.www.carcano.ch`.

Once the DNS TXT record propagates, just press the Enter key: in a while `certbot` retrieves the certificate and prints several messages.

The messages report

- The name of the file containing the issued certificate: it is the one specified by the line starting with `Certificate is saved at`, so the file name is `0000_cert.pem`.
- The name of the file containing the CA's chain certificate(s), which are the ones used to sign the above-issued certificate. This information is provided by the line starting with `Intermediate CA chain is saved at`, so the file name is `0000_chain.pem`.
- The name of the file containing the certificate bundles with both the issued certificate and the CA's chain certificate(s). This is the one reported by the line starting with `Full certificate chain is saved at`, so the file name is `0001_chain.pem`. We will discuss in detail when it is necessary to use this kind of bundle in Chapter 25 when talking about Transport Layer Security (TLS).

As you can see, `certbot` also warns you about when the certificate will expire and informs you that you are supposed to manually request the renewal as necessary before its expiration.

```
This certificate expires on 2025-05-21.
```

Note Here we have run `certbot` in manual mode only for didactic purposes: the proper way to use it would be to have the whole certificate process managed by it, for example, by running `certbot --nginx -d carcano.ch -d www.carcano.ch`. In this specific example, `certbot` "automatically" configures the NGINx server for using the issued certificate. When running `certbot` in this mode, remember also to create a daily `Systemd` timer or a `cron` job with the statement `/usr/bin/certbot renew --quiet` so as to have `certbot` automatically renew the certificate before its expiration.

Let's have a look at the issued certificate:

```
openssl x509 -noout -text -in 0000_cert.pem
```

As you can see, it contains the same data as the CSR:

```
Subject: CN = www.carcano.ch
```

```
Subject Public Key Info:
    Public Key Algorithm: id-ecPublicKey
        Public-Key: (384 bit)
        pub:
            04:82:62:ee:01:43:bf:28:42:1b:27:ec:01:87:33:
        ASN1 OID: secp384r1
        NIST CURVE: P-384
```

Although we have not explicitly asked for that, some data are also added to the `Subject Alternative Name` extension:

```
X509v3 extensions:
    X509v3 Subject Alternative Name:
        DNS:www.carcano.ch
```

Note Notably, the CA/Browser Forum (CAB Forum) rules mandate that the value of the CN key must also be included among the SANs for publicly trusted certificates.

Please note also how the certificate provides a clear statement about the certificate's constraints:

```
X509v3 Basic Constraints: critical
    CA:FALSE
```

The term `critical` following `X509v3 Basic Constraints` indicates that this extension must be interpreted according to the claims that follow. In this instance, the only claim is `CA: FALSE` which means that this certificate cannot be used as a CA certificate under any circumstances. Therefore, it must always be interpreted as a leaf (end-entity) certificate.

This certificate provides also information about the intended key usages:

```
X509v3 Key Usage: critical
  Digital Signature
X509v3 Extended Key Usage:
  TLS Web Server Authentication, TLS Web Client Authentication
```

Because of the presence of the `critical` following `X509v3 Key Usage`, this extension must also be strictly interpreted. From this information, we understand that the permitted usages for this certificate are TLS server and client authentication, as well as generating digital signatures.

In addition to that, the certificate also contains PKI information that can be used for certificate chain validation.

The most important ones are the digital signature generated using the issuer's certificate (the details are truncated to save space):

```
    Signature Algorithm: ecdsa-with-SHA384
    Signature Value:
        30:66:02:31:00:f3:90:4e:55:2e:6c:d5:03:c9:9c:74:79:87:
```

and the `Issuer` field, since it is used to identify the signer certificate during the certificate chain validation process:

```
Issuer: C = US, O = Let's Encrypt, CN = E6
```

Other important bits are the certificate lifetime:

```
Validity
    Not Before: Feb 20 17:35:05 2025 GMT
    Not After : May 21 17:35:04 2025 GMT
```

and the Authority Key Identifier.

```
X509v3 Authority Key Identifier:
    93:27:46:98:03:A9:51:68:8E:98:D6:C4:42:48:DB:
```

Note Both the issuer and the Authority Key Identifier are essential for validating the certificate chain. The issuer is used to locate the Certificate Authorities (CAs) that signed the certificate, while the Authority Key Identifier is used to identify the specific CA certificate used for signing.

Another very important piece of PKI information provided by the certificate is the Authority Information Access (AIA) extension:

```
Authority Information Access:
    OCSP - URI:http://e6.o.lencr.org
    CA Issuers - URI:http://e6.i.lencr.org/
```

Note Some months after this chapter was written, let's Encrypt decommissioned its OCSP service and switched to CRL. Therefore, OCSP information is no longer available in the Authority Information Access extension.

This extension provides the CA issuer's URI, which is the URL used to download the intermediate CA certificates.

Additionally, it can be used to provide information about the endpoints that can be used to query the Online Certificate Status Protocol (OCSP) responder, enabling clients to check the revocation status of the certificate in real-time.

However, it's important to remark that not all clients can automatically utilize this AIA information to retrieve the intermediate certificates. For instance, it is quite rare for web service clients to do so.

Note After Let's Encrypt OCSP service was decommissioned, the status of their certificates can be checked using Certificate Revocation List (CRL): the CRL URI can be found in the CRL Distribution Points (CDP) x509v3 extension.

Validating the Certificate

Since the certificate is signed by a publicly recognized Certificate Authority, then the certificate is supposed to be trusted.

As an exercise, let's make sure of this by using the `openssl` command-line tool:

```
openssl verify 0000_cert.pem
```

The output may look discouraging:

```
error 20 at 0 depth lookup: unable to get local issuer certificate
error 0000_cert.pem: verification failed
```

That answer anyway is correct: `openssl` is just complaining that it was unable to assemble the certificate chain to perform the validation. This is caused by missing the intermediate certificates in the `0000_cert.pem` file.

Assembling the Certificate Chain

As we already said, assembling the certificate chain requires identifying the issuer of each certificate in the chain and seeking the CA certificate having that subject.

As an exercise, let's manually do this process.

First, let's check the certificate's issuer as follows:

```
openssl x509 -in 0000_cert.pem -noout -issuer
```

The output is

```
Issuer: C = US, O = Let's Encrypt, CN = E6
```

This means we must locate the file containing the CA certificate having `C = US, O = Let's Encrypt, CN = E6` as subject.

If we look among the files provided by `certbot`, there is also the `0000_chain.pem` file.

Let's check its subject and issuer:

```
openssl x509 -in 0000_chain.pem -noout -subject -issuer
```

The output is

```
subject=C = US, O = Let's Encrypt, CN = E6
issuer=C = US, O = Internet Security Research Group, CN = ISRG Root X1
```

Since the subject matches the issuer of the `0000_cert.pem` certificate, then the `0000_chain.pem` contains the CA certificate used to sign the TLS server certificate `0000_cert.pem` we are looking for.

In addition to that, since its subject and issuer differs, then this is an intermediate CA certificate.

So, to summarize, we have the following certificate chain:

```
CN=ISRG Root X1
     \----CN = E6 (untrusted CA)
              \------ CN = www.carcano.ch (unverified cert)
```

As you can see from the above graph, the `CN = E6` intermediate CA certificate, stored in the `0000_chain.pem` file, is untrusted. This means that, to have openssl correctly validate the chain, we must use the `-untrusted` parameter to tell `openssl` to include the `0000_chain.pem` intermediate certificate in the validation process.

As for the `ISRG Root X1` CA certificate, we can argue from its name it is a ROOT CA: since Let's Encrypt is a globally recognized public Certificate Authority, we also expect it to be already contained in the system-wide truststore.

We now have all the necessary information to attempt the certificate chain validation. Just run

```
openssl verify -untrusted 0000_chain.pem 0000_cert.pem
```

Indeed, the output of the `openssl verify` command this time is

```
0000_cert.pem: OK
```

Note In this scenario `0000_chain.pem` contains only one intermediate CA certificate, but some globally recognized public Certificate Authorities, such as GoDaddy, use more than one intermediate CA certificate chained together. In such a scenario, each single intermediate CA certificate must be bundled within the file passed with the `-untrusted` option to enable the chain to be completed.

So, when dealing with CA-signed certificates, unless they are signed by a ROOT CA, we still have the problem of passing one or more certificates to the client – intermediate CA certificates in this case.

In Chapter 25, when talking about TLS, we will see how TLS addresses this by having the TLS server pass to the client not only the TLS server's entity certificate itself, but also the bundle with all the intermediate CA certificates necessary to assemble the chain.

Certificate Status

As we said the lifetime of a certificate is defined by a specific range of dates, so, when validating the certificate, the current date is verified to ensure it falls within this range.

It is important to know that having a fixed lifetime for certificates presents the significant security risk of being unable to react effectively if a certificate's private key is compromised before the certificate's expiration. This situation allows an attacker to use the compromised key to impersonate the certificate holder until the certificate's expiration date is reached.

The Public Key Infrastructure (PKI) offers two primary solutions to this problem: Certificate Revocation Lists (CRLs) and the Online Certificate Status Protocol (OCSP).

A CRL is an online-published list of revoked certificates that is signed by the issuing Certificate Authority (CA).

In contrast, OCSP is a client-server protocol that enables clients to query the status of specific certificates in real-time.

Online Certificate Status Protocol (OCSP)

Note Some months after this chapter was written, Let's Encrypt dismissed its OCSP service and switched to CRL. Therefore, OCSP information is no longer available in the Authority Information Access extension – `https://letsencrypt.org/2025/08/06/ocsp-service-has-reached-end-of-life`. I decided to keep this paragraph anyway since it still provides a good explanation of how OCSP works and how to operate when dealing with certificates supporting it – just read it without running the examples.

As we just said, OCSP is a client-server protocol enabling the client to query the status of specific certificates on the fly: this approach addresses the shortcomings of CRLs, which require each client to download the entire list of revoked certificates. As the list grows, especially for large CAs, this shortcoming can lead to performance issues.

Additionally, the OCSP protocol allows applications to prefetch OCSP responses for their certificates and provide these responses stapled along with the certificates. This technique, known as OCSP stapling, spares the client the time required to query the OCSP responder and reduces the number of client connections to the OCSP responder.

For these reasons, most of the modern Certificate Authorities prefer to implement OCSP over CRL, despite some CAs still providing CRL to support very old clients.

To better understand how OCSP works, let's see an OCSP check in action.

Since OCSP replies are signed by the CA which signed the certificate you are checking (or a delegated one, but the standard is quite flaky on this topic), we must first download that intermediate CA certificate into a file.

As we saw, in our example `certboot` already provided it in the `0000_chain.pem` file, so we can skip the downloading part.

The check statement requires us to provide the OCSP endpoint URI. This information can be easily obtained from the certificate we want to check by running

```
openssl x509 -noout -ocsp_uri -in 0000_cert.pem
```

In our example the answer is

```
http://e6.o.lencr.org
```

We can now run an OCSP query to check the status of the `www.carcano.ch` certificate by running

```
openssl ocsp -issuer 0000_chain.pem \
-url http://e6.o.lencr.org -cert 0000_cert.pem
```

If the certificate has not been revoked, the answer look as follows:

```
WARNING: no nonce in response
Response verify OK
0000_cert.pem: good
     This Update: Feb 24 20:18:00 2025 GMT
     Next Update: Mar  3 20:17:58 2025 GMT
```

Note This warning indicates that the OCSP response did not include a nonce: a nonce is a unique identifier that helps to prevent replay attacks. In this case the OCSP server is not sending the nonce: while this absence may not necessarily indicate a problem, it is generally recommended to include one for added security.

An answer like

```
Response verify OK
0000_cert.pem: revoked
```

Means instead that the certificate was revoked.

Certificate Revocation List (CRL)

A Certificate Revocation List (CRL) is a comprehensive list of certificates that have been revoked before their scheduled expiration date. This list is periodically regenerated and published online, allowing clients to easily download and check for revoked certificates.

A CRL must be signed by the same Certificate Authority (CA) that issued the certificates it lists, or it can be signed by a delegated CA certificate in cases where an indirect CRL is used.

Note Be aware that not all TLS clients correctly implement validation for indirect CRLs, which can lead to potential security issues.

If a CA utilizes CRLs, the best practice is to provide the URL from which to download it by adding the CRL Distribution Point (CDP) extension of the X.509 certificate. Anyway, mind that this is not a mandatory requirement.

A good candidate for running a quick test are Google's certificates, since they contain the CDP extension.

We can use the `retrieve-certs.sh` in this chapter's workspace to connect to https://www.google.com and extract every certificate from the bundle into a distinct file.

Just run

```
./retrieve-certs.sh www.google.com
```

While processing the chain, it prints its structure as follows:

```
depth=2 C = US, O = Google Trust Services LLC, CN = GTS Root R1
verify return:1
depth=1 C = US, O = Google Trust Services, CN = WR2
verify return:1
depth=0 CN = www.google.com
verify return:1
DONE
```

```
File www.google.com.crt: subject=CN = www.google.com
File WR2.crt: subject=C = US, O = Google Trust Services, CN = WR2
File GTS-Root-R1.crt: subject=C = US, O = Google Trust Services LLC,
CN = GTS Root R1
Intermediate CA certificates saved as ca-chain.pem
```

Let's first ensure that the www.google.com certificate provides the CRL Distribution Point (CDP):

```
openssl x509 -noout -text -in www.google.com.crt \
| grep -A 2 "CRL Distribution Points"
```

The output is

```
X509v3 CRL Distribution Points:
    Full Name:
        URI:http://c.pki.goog/wr2/oBFYYahzgVI.crl
```

CRL verification is invoked by passing the `-crl_check` option of the `openssl verify` statement: since the certificate we want to verify provides the CDP, we specify the `-crl_download` option so to ask `openssl` to automatically download the CRL using the URI provided by the extension.

As we already saw, it is also necessary to use the `-untrusted` option followed by the path to the intermediate CA bundle or `openssl` won't be able to assemble the certificates chain.

The full statement is as follows:

```
openssl verify -untrusted ca-chain.pem -crl_check \
-crl_download www.google.com.crt
```

In the event the certificate we want to check does not provide the CDP, we can ask the Certificate Authority's support for the URL for downloading the CRL. Once obtained, we can download it and use the `-CRLfile` option followed by the CRL file downloaded.

Keystores

The PEM-encoded private key and certificates we created so far were always stored in distinct files, with the private key encoded as necessary.

Anyway, some applications require having both the private key and the certificate within the same file. These files are often called a certificate bundle or Keystore or Keystore Containers.

PEM Containers

This is the most trivial container format: the PEM-encoded certificate and private key are simply concatenated within the same file.

For example:

```
cat wildcard.t1-t3.carcano.corp.crt \
wildcard.t1-t3.carcano.corp.key > \
wildcard.t1-t3.carcano.corp.pem
```

Note If the certificate is signed by an intermediate CA, it is necessary to concatenate also the whole intermediate CA certificates bundle.

Applications supporting this bundle format can now load just the `wildcard.t1-t3.carcano.corp.pem` file, since it contains everything necessary to operate.

JKS Containers

When dealing with older Java applications, it is common for them to require the use of Java KeyStore (JKS)-formatted containers.

JKS is a secure container specifically designed for Java applications to store cryptographic keys, certificates, and trusted certificate authorities. Its security model mandates that you provide at private key generation time the components of the certificate subject (such as `CN`, `O`, and so on). To enhance the protection of sensitive key material, its security model forbids the direct export of private keys.

Note Mind that it is possible to export the private key by converting the JKS into a PKCS12-encoded container, which instead allows for the export of private keys along with their associated certificates.

JKS-formatted containers are generated and managed using the `keytool` command-line tool.

Note Since `keytool` is a Java-related tool, to work with the JKS format, it is necessary to have Java installed on the system. If it isn't, just run `sudo dnf install -y java`.

As an example, let's generate the `certs.jks` JKS container with a 2048-bit RSA key for the "CN=www.p1.carcano.corp, O=Carcano SA, C=CH" distinguished name:

```
keytool -storetype JKS -keystore certs.jks -genkey \
-alias "www.carcano.corp" -keyalg RSA -keysize 2048 \
-dname "CN=www.p1.carcano.corp, O=Carcano SA, C=CH"
```

It will ask you twice to create a password: the first is for the JKS container itself, the second is specific for the "www.carcano.corp" private key. Enter it just the first time (repeating it twice); then, as suggested, press enter to reuse it for the private key too.

Note Although it is technically possible to set different passwords for the container itself and for the private key, in my experience there are a lot of bugged libraries around that cannot handle them when they are different. To maximize portability, my personal advice is to always have both the JKS container and the certificate use the same password.

Since the JKS container also contains the private key, it is not possible to directly submit it to a CA. It is instead necessary to export a PEM-formatted CSR as follows:

```
keytool -keystore certs.jks -certreq \
-alias "www.carcano.corp" -file certs.csr
```

The `certs.csr` is a regular CSR – we can indeed inspect it using `openssl` as follows:

```
openssl req -in certs.csr -noout -text
```

Continuing this fictional example, let's suppose we now submit the `certs.csr` to the `Carcano PI2 CA` private CA. At the end of the process, we have two distinct PEM-formatted files:

- `cert.crt` containing the issued certificate
- `carcano-pi2.crt` containing the intermediate certificate bundle

To complete the process, we must import both files into the JKS container.

First, we import the CA's bundle as follows:

```
keytool -keystore certs.jks -noprompt \
-importcert -alias "Carcano PI2 CA" -file carcano-pi2.crt
```

Then we import the issued certificate:

```
keytool -keystore certs.jks -noprompt \
-importcert -alias "www.carcano.corp" -file cert.crt
```

Any application supporting the JKS container can now load just the `certs.jks` file, since it contains everything necessary to operate.

Note If necessary, you can list the contents of a Keystore by running `keytool` with the `-list` action. For example: `keytool -keystore certs.jks -list`. The official online documentation of the `keytool` command is available on Oracle's Help Center.

PKCS#12 Containers

PKCS#12 is an IETF standard developed to provide a standardized format for securely storing private keys and certificates in a single password-protected container file, promoting interoperability across different platforms and applications.

Due to its industry-standard status, enhanced security features, and compatibility with various cryptographic algorithms, it gradually became a more versatile choice for Keystores replacing JKS.

Note Concerning FIPS 140-2 level II, both JKS and PKCS#12 are suitable containers, but they must be used in conjunction with FIPS-approved cryptographic algorithms and libraries. When dealing with Java, this means that the Java Cryptography Architecture (JCA) and the underlying cryptographic providers must be configured to use FIPS-compliant algorithms.

The `keytool` command-line utility supports also the `PKCS#12` format. The statement to use is the same: the only difference is that in this case it is necessary to specify `PKCS12` as the value of the `-storetype` option.

Note The PKCS12 format is supported also by the `openssl` command-line tool. As an exercise, I suggest you re-try with `openssl` the same exercises we did with `keytool`. As a reference, you may use the official documentation available at https://docs.openssl.org/1.1.1/man1/pkcs12.

Private CA

As we said, when dealing with private CA's, if we want to have them trusted the same way as the globally available ones, we must trust their ROOT certificates by adding them to the system-wide truststore.

On Red Hat family distributions, this can be easily achieved by adding the file containing the ROOT CA certificate to the `/etc/pki/ca-trust/source/anchors` directory – this obviously requires administrative privileges.

Once added the certificates you want to trust, simply run

```
sudo update-ca-trust extract
```

Application-Specific Truststores

Sometimes you may need to restrict access to specific applications only to certificates issued by specific CAs. In such a scenario, the application is typically launched using a command-line option or a setting in the configuration file which provides the path to a custom truststore file containing only the ROOT CA certificates you want to trust. This prevents certificates signed by any other CAs, including the globally available ones, from passing the CA-chain validation.

PEM Containers

The most trivial way of creating an application-specific truststore is concatenating the ROOT CA's PEM-encoded certificates you want to trust into a single truststore file.

We just need to concatenate them using the `cat` command-line utility redirecting the output as follows.

As an example, let's suppose we want to create a custom truststore with the ROOT CA certificates of the `CarcanoROOT-P1.crt` and `CarcanoROOT-P2.crt` files:

```
cat CarcanoROOT-P1.crt CarcanoROOT-P2.crt > truststore.pem
```

The application can now load only the `truststore.pem`, since it contains all the needed ROOT CA certificates.

Note Be wary that the best practice is not to put intermediate CA certificates in the truststore: always put only ROOT CA certificates.

The drawback of this format is that, since the truststore just contains the PEM-encoded certificates, it does not provide an easy way to infer which CA certificates are stored in it.

A more elegant and tidy way to deal with it, which also addresses this problem, is to use the `openssl` statement to add each certificate's subject and issuer right before the certificate text.

For example:

```
openssl x509 -in CarcanoROOT-P1.crt -subject -issuer \
> truststore.pem
openssl x509 -in CarcanoROOT-P2.crt -subject -issuer \
>> truststore.pem
```

This simplifies inferring which certificates are stored in the truststore as well as locating the certificate text of specific ones.

Sadly, you will quite often (if not always) find truststores generated by simply concatenating CA certificates using `cat`. When dealing with truststores created in this way, a quite common frustrating situation is seeking a specific certificate having a certain subject.

In such a situation, you can use the trick of converting the PEM-encoded truststore to PKCS7 on the fly and pipe the output again to openssl, this time asking it to print its contents:

```
openssl crl2pkcs7 -nocrl -certfile ca-chain.pem \
| openssl pkcs7 -print_certs
```

The output of the above statement is the list of concatenated PEM-encoded certificates, having each of them prepended by both subject and issuer.

As you can see this is a trick exploiting an action that is supposed to be used with CRLs: so, mind that since it is an improper use of the tool, there is no guarantee it will always work in the future.

JKS Containers

When dealing with old Java applications, we can of course use JKS containers as a truststore: in such a scenario, we simply skip the key generation part, going straight to the import certificate part of the process.

As an example, suppose we need to import the `Carcano Root P1` CA certificate into a JKS truststore.

The statement to run is as follows:

```
keytool -storetype JKS -keystore truststore.jks -noprompt \
-importcert -alias "Carcano Root P1" -file CarcanoROOT-P1.crt
```

When prompted, type the password you want to assign to the truststore.

Similarly, we can import other ROOT CA certificates into the same truststore. For example:

```
keytool -keystore truststore.jks -noprompt \
-importcert -alias "Carcano Root P2" -file CarcanoROOT-P2.crt
```

When requested, type the password for opening the truststore.

We can then list the certificates contained in the truststore by running

```
keytool -list -keystore truststore.jks
```

If necessary, we can remove specific certificates from the truststore.

For example, to remove the `Carcano Root P2` certificate, run

```
keytool -keystore truststore.jks -delete -alias "Carcano Root P2"
```

PKCS12 Containers

The statements are almost the same: it differs only when creating the container, since in this case you must specify `PKCS12` as the value of the `-storetype` option.

Summary

In this chapter, we delved into X.509 certificates and Public Key Infrastructure (PKI), not only in theory but also through practical exercises using OpenSSL, along with a few BASH and Python scripts. We began with self-signed certificates, learning the principles of certificate validation. As we progressed, we enhanced our skills by understanding the advantages of using signed certificates and key concepts such as Certificate Authorities (CA), CA-chain validation, and truststores.

We also explored advanced topics, including Certificate Revocation Lists (CRL) and Online Certificate Status Protocol (OCSP) validation.

For DevOps and DevSecOps professionals, mastering X.509 certificates and PKI is crucial for establishing secure communications and ensuring the integrity of data exchanges. These skills enable the implementation of secure SSL/TLS connections, which are essential for protecting sensitive information. Additionally, understanding certificate management and validation processes helps to mitigate risks associated with unauthorized access and data breaches. By effectively utilizing X.509 certificates and PKI, professionals can foster trust in their systems and ensure compliance with security standards, ultimately contributing to a more secure development lifecycle.

PART VIII

Common Protocols

Applications exchange data using protocols: for this reason, having a good understanding of these protocols is an essential skill for every IT professional. This is particularly true for DevOps and DevSecOps professionals, given their intermediary position between development, QA, and operations teams. A solid understanding of protocols can significantly enhance a professional's role, transforming them from just another team member into a point of reference for colleagues across various teams.

The chapters in this section delve into the following commonly used protocols:

- **HTTP**: The most broadly used application-layer protocol, functioning as the primary vehicle for encapsulating and transporting modern service architectures such as REST, SOAP, and web payloads.
- **LDAP**: A binary, ASN.1-encoded application protocol implemented through text-based syntax models—such as Distinguished Names (DN) and LDIF structures—widely deployed for enterprise directory services and centralized identity authentication.
- **TLS**: A cryptographic encapsulation protocol designed to wrap application-layer transport streams—such as HTTP and LDAP—enforcing asymmetric identity verification, mutual trust boundaries, and end-to-end data confidentiality across untrusted networks.

Skills in these three common protocols represent the minimum expected protocol skill set for professionals, as most solutions running on the Internet and within corporate intranets rely on them.

CHAPTER 24

HTTP

HTTP is certainly the most widely used text-based transport protocol: it serves as a foundational layer for various service protocols, including REST and SOAP, as well as applications like Git. For DevOps and DevSecOps professionals, mastering HTTP is crucial for effectively supporting development and quality assurance (QA) teams.

A solid grasp of HTTP status codes, the ability to utilize and exploit HTTP headers, and knowledge of content type negotiation and caching mechanisms are vital skills. These competencies not only facilitate troubleshooting of complex issues that may arise during the software development lifecycle but also contribute to sound software design practices.

Proficiency in HTTP is indispensable for professionals as it enhances collaboration between development and security teams, streamlines the troubleshooting process, and fosters the creation of robust, secure applications.

The Hypertext Transfer Protocol

It was in 1989 when Tim Berners-Lee, while he was at CERN, defined the specification of a **clear-text** application-layer protocol suitable for transmitting hypertext over the Internet.

That protocol was called Hypertext Transfer Protocol, broadly known by nearly everybody as HTTP.

The first version, HTTP/0.9, released in 1991, was reviewed and improved several times over the years (HTTP/1.0 in 1996, HTTP/1.1 in 1999) until HTTP/2, the current version published in 2015. The HTTP/3 revision is instead currently being developed.

The primary purpose of HTTP is to enable the transfer of data over the web, allowing users to request and receive resources such as HTML documents, images, and multimedia contents.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_24

Note Be wary that HTTP is a clear-text protocol so, whenever confidentiality is a requirement, you must encapsulate it using Transport Layer Security (TLS) – in turn: use HTTPS: we'll discuss TLS in the next chapter.

The HTTP specifications define two types of messages:

- Request messages
- Response messages

Request Messages

These are the messages sent from clients to servers to request resources or to perform actions as summarized in Table 24-1.

Table 24-1. *HTTP Request actions*

Method	Idempotent	Description
HEAD	Yes	Requests only the headers of a resource, without the body. It is often used to check if a resource exists (e.g., as a health check) or to retrieve metadata. It is considered a safe method, since it does not send data to the server.
OPTIONS	Yes	Requests to list the communication options for the target resource: it is typically used from the client to guess which HTTP methods are supported for a specific resource.
GET	Yes	Retrieves data from the server. Mind that, to improve performance, responses can be cached by browsers and intermediate proxies. It is considered a safe method, since it does not send data to the server.
POST	No	Sends data to the server to create or update a resource (e.g., form submissions): the actual resource contents are in the body of the HTTP message.
PUT	Yes	Updates an existing resource by entirely replacing it or creates it if it does not exist: the actual resource contents are in the body of the HTTP message.
PATCH	No	Applies partial modifications to a resource, updating only the specified fields. Depending on the implementation, sending the same PATCH request multiple times may yield different results.
DELETE	Yes	Removes a resource from the server.

Note Idempotent means that multiple identical requests always produce the same outcome as a single request.

The following example shows the structure of an HTTP request message:

```
POST /api/users HTTP/1.1
Host: foo.carcano.corp
User-Agent: Mozilla/5.0
Accept: application/json
Content-Type: application/json
Content-Length: 56

[
  {
    "name": "Marco Antonio Carcano",
    "email": "marco.carcano@carcano.corp"
  }
]
```

It is made of several components:

- Request line (line 1)
- Headers (lines 2–6)
- Blank line before the body (line 7)
- Body (lines 8–13)

Table 24-2 describes them a little more in detail.

Table 24-2. *HTTP request message structure*

Item	Description
Request line	Describes the request, specifying the accessed resource and the access method. The syntax is `METHOD URI HTTP/VERSION`
Headers	Colon-separated key-value pairs, one per line, used to provide additional information about the request. They are often called `Request Headers`.
Blank Line	A blank line necessary for separating the headers from the body.
Body	Optional contents with the data being sent to the server, typically used with methods like POST or PUT or PATCH.

Table 24-3 summarizes the most commonly used request headers.

Table 24-3. *HTTP most commonly used request headers*

Component	Description
Host	It specifies the destination's HTTP Host's FQDN or IP address to which the request is being sent. Optionally, it can also contain the port number separated by a colon. Be wary that despite this header being optional in HTTP/1.0, it is mandatory for HTTP/1.1. For example: `Host: foo.carcano.corp`
User-Agent	Describes the client application generating the request, including the browser type and version, operating system, and device type. It is used to enable the servers to tailor responses based on the client's capabilities, such as serving different content for mobiles or desktops.

(*continued*)

Table 24-3. (*continued*)

Component	Description
Accept	It is related to content negotiation, specifying the media types that the client is willing to receive in the response. For example: `Accept: application/json, text/html;q=0.9, */*;q=0.8` The q is the weight for the preference. In the above example, `application/json` is the preferred format, immediately followed by `text/html`, failing over the other contents (`*/*`) if both formats are not available from the server.
Accept-Encoding	The content encoding (compression) method supported by the client: it is used to save bandwidth by compressing the response. For example: `Accept-Encoding: gzip, deflate, br`
Accept-Language	The preferred language for the response: it is used to enable the server to provide localized content based on the client's preferences. For example: `Accept-Language: en-US,en;q=0.9,fr;q=0.8`
Content-Type	Tells the server which media type is contained in the request body: it is nearly mandatory in POST, PATCH, and PUT requests. For example: `Content-Type: application/json`
Content-Length	Specifies the length in bytes of the request body, informing the server of the expected size of the object to receive. For example: `Content-Length: 123` This helps in detecting mismatches.
Cookie	Sends back to the server previously stored cookies received from it. Cookies are information stored on the client side, which enables the server to maintain state and track user sessions. For example, if the server is a reverse HTTP proxy, it may use a previously set cookie to track which backend served the initial client's request to always send every next request to the same backend node. This enables stateful sessions when dealing with replicated services. For example: `Cookie: backend=foo-01.carcano.corp`

(*continued*)

Table 24-3. (*continued*)

Component	Description
Authorization	Provides the credentials for authenticating the client with the server. For example: `Authorization: Basic dXNlc6cc3dvcmQxMjM=` There are other authentication schemes: the above example uses the `Basic` scheme. This by the way is the less confidential one, since it just base64 encodes the credentials. More secure methods are `Digest` and `Bearer`.

Response Messages

These are the messages sent by servers back to clients in response to their requests. Besides giving back the requested contents (or an error message if it is not found), they also indicate the outcome of the request.

The following example shows the structure of an HTTP response message:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 123
Date: Wed, 13 Nov 2024 07:28:00 GMT

{
  "id": 1,
  "name": "Marco Antonio Carcano",
  "email": "marco.carcano@carcano.corp"
}
```

It contains several components

- Status line (line 1)
- Headers (line 2–4)
- Blank line before the body (line 5)
- Body (lines 6–10)

Table 24-4 describes them a little more in detail.

Table 24-4. *HTTP response message structure*

Item	Description
Status line	A summary of the response with the following format `HTTP/VERSION STATUS_CODE REASON_PHRASE` where: • `HTTP/VERSION` is the HTTP version being used • `STATUS_CODE` is a three-digit code indicating the result of the request (e.g., 200, 404, 500). • `REASON_PHRASE` is a brief description of the status code, such as OK, `Not Found` and so on.
Headers	Colon-separated key-value pairs, one per line, used to provide additional information about the request and caching directives. They are often called `Response Headers`.
Blank Line	A blank line necessary for separating the headers from the body.
Body	Optional contents with the data being sent to the client.

The outcome of a request is reported by using HTTP status codes.
HTTP status codes are grouped into classes as summarized in Table 24-5:

Table 24-5. *HTTP status classes*

Item	Class	Description
1xx	Informational	Request received, continuing process. For example: `100 Continue`.
2xx	Success	The request was successfully received, understood, and accepted. For example: `200 OK` or `204 No Content`.
3xx	Redirection	Further action must be taken to complete the request. For example: `301 Moved Permanently` or `302 Found`.
4xx	Client Error	The request contains bad syntax or cannot be fulfilled. For example: `404 Not Found` or `403 Forbidden`.
5xx	Server Error	The server failed to fulfill a valid request. For example: `500 Internal Server Error` or `503 Service Unavailable`.

Note IANA maintains the official full list of HTTP status codes. It is available at https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml.

Table 24-6 summarizes the most commonly used response headers.

Table 24-6. *HTTP most commonly used response headers*

Component	Description
Content-Type	The same as the request header, but this time in the context of a server response.
Content-Length	The same as the request header, but this time in the context of a server response.
Date	Time when the response was generated – useful for tracking and logging. For example: `Date: Wed, 13 Nov 2024 07:28:00 GMT`
Server	Information about the server software handling the request. For example: `Server: nginx/1.23.0` Mind that in publicly available services, the best practice is to suppress this header to prevent security information leaks.
Cache-Control	Controls how the response can be cached by browsers and intermediate caches. For example: `Cache-Control: no-cache, no-store, must-revalidate`
Expires	Used to suggest to the client how long the response can be cached. For example: `Expires: Wed, 20 Nov 2024 07:28:00 GMT`

(*continued*)

Table 24-6. (*continued*)

Component	Description
Location	Redirect the client to another location – typically used with `3xx` status codes. For example: `Location: https://foo.carcano.corp` A very typical usage, to promote security, is redirecting HTTP requests to the same URL, but using the HTTPS protocol.
Content-Encoding	Indicates the encoding used on the response body (e.g., compression), allowing the client to decode the response body using the proper algorithm. For example: `Content-Encoding: gzip`
Content-Disposition	Suggests to the client how the downloaded content should be displayed. For example, if the downloaded document is supposed to be directly rendered or saved as an attachment. An example of this header is `Content-Disposition: attachment; filename="file.pdf"`
Set-Cookie	Sends session information to be stored on the client side, such as session identifiers. For example: `Set-Cookie: backend=foo-01.carcano.corp; Path=/; HttpOnly; Secure`
Access-Control-Allow-Origin	Specifies which origins are allowed to access the resource in Cross-Origin Resource Sharing (CORS), enhancing security for web applications. For example: `Access-Control-Allow-Origin: *`

HTTP Virtual Host

One of the benefits of the `Host` HTTP header is that it enables virtual hosting. This feature allows multiple virtual HTTP hosts to share the same IP address, with the server selecting the appropriate one by inspecting the `Host` header.

This capability is widely used, as it significantly reduces the need for multiple IP addresses and minimizes overall configuration and maintenance efforts. Nearly every current HTTP service implements this feature.

HTTP Proxies

It is possible to have an HTTP service that instead of directly serving resources acts as an intermediary between a client and a server: these are called HTTP proxies.

Note HTTPS proxies can be used also for terminating TLS connections, enabling the inspection of the decrypted payload: we will discuss TLS in the chapter dedicated to security. For now just mind they can be used this way for auditing and tracking purposes.

From a bird's-eye view, HTTP proxies have basically only two macro use cases: handling incoming requests (reverse proxies) or handling outgoing requests (outgoing proxies).

Reverse Proxies

Reverse Proxies are HTTP Proxies placed in front of the service endpoints. They address the following purposes:

- Load balancing, spreading requests on multiple backends.
- Request routing, forwarding requests to the proper resource endpoints. In some scenarios they are also called API gateway or, in micro-services terminology, application routers.

In both cases, they can also provide caching for static contents, and, if they have Web Application Filtering (WAF) capabilities, they can also inspect payload dropping malicious requests.

Note It is quite common having reverse proxies chained to implement multiple security zones, for example, having reverse proxies with WAF capabilities in the demilitarized zone (DMZ) and internal proxies used only for load-balancing.

A huge benefit of reverse proxies is enabling end-to-end request tracking: this means being able to monitor and log each single request as it moves through various layers of an application stack, from the HTTP layer through the application logic and database interactions (such as SQL queries).

This is achieved by having the outermost HTTP proxy insert an HTTP header with a unique identifier: every application managing the request will then add that identifier into the logs, enabling log aggregators to use it as the log correlation identifier. Commonly used names for such a header are `X-Request-ID`, `X-Correlation-ID`, and `X-Trace-ID`.

The drawback with reverse proxies is that they hide the requestor's IP address to the target server; to overcome this shortcoming, the best practice is to have the most outer proxy setting the `X-Forwarded-For` HTTP header containing the requestor's IP address: this enable the backends to correctly track the requestor's IP address in their logs as well as to implement access controls or rate limiting policies based on the original client's IP as necessary.

Outgoing Proxies

Outgoing proxies are HTTP proxies installed to enable clients on networks isolated from the Internet to reach HTTP or HTTPS services in the outside world, or to reach services belonging to a different security tier homed into other isolated networks.

Besides the benefit of providing a local cache of remote contents, they are often used not only to track requests, but also to block access to certain websites or contents.

Note In high-security environments or those subject to regulations, it's common to have a default rule that denies access to any destination, requiring clearance for approval. Sometimes, these clearance rules are so strict that exceptions must be granted at a granular level, such as for a single client to a specific destination.

If the HTTP proxy has Web Application Filtering capabilities, it can also be used for Data Leak Prevention (DLP) purposes or to block traffic only of specific applications – for example, deny the use of social networks or to block malwares.

Note Internet providers often use HTTP proxies to cache contents in locations close to their customers. This is often called content acceleration. Another use case for remote HTTP proxies is bypassing geo-restrictions, exploiting the fact the request will look like it is being generated from the proxy's IP address.

To be able to use a proxy, client applications must be configured at least with the URLs of the HTTP proxy endpoint, but it may be necessary to specify also the port and the credentials for authenticating to it.

On Linux, most of the applications read the HTTP proxy settings from a standard set of environment variables:

- `http_proxy`: HTTP's proxy URL
- `https_proxy`: HTTPS' proxy URL
- `no_proxy`: A list of URLs that must avoid using the HTTP or HTTPS proxy – a typical value must include `localhost`, `127.0.0.1`, and every reachable network

Anyway, some applications, such as the `docker` command-line tool or Java applications, use their own way for setting the HTTP proxy to be used, such as a configuration file, properties, or a command-line option.

Note Although it is technically possible to include username and password to authenticate to the proxy in these values, this is a very old and insecure practice that must be avoided.

For example, to have these variables automatically set for every user logged into the system, add them to the `/etc/profiles.d/proxy.sh` (create it if it does not exist). For example:

```
export http_proxy="http://proxy.carcano.corp"
export https_proxy="https://proxy.carcano.corp"
export no_proxy="localhost,127.0.0.1,.carcano.corp"
```

Note Don't store any credentials in this file, unless your corporate security policy allows you to disclose and share them with other colleagues: everyone logging on the system with enough rights may be able to see them indeed.

If these variables have not been configured to be automatically set, they must be manually exported each time it is necessary to perform connections going through the proxy.

Note Mind that some networks are configured to enable transparent proxies: these are implemented by having a firewall rule on the default gateway forwarding all the TCP port 80 traffic to the HTTP proxy, and all the TCP port 443 traffic to the HTTPS proxy.

Summary

In this brief chapter, we provided a comprehensive overview of the HTTP protocol.

We explored HTTP methods and status codes, delving into both request and response messages, and discussed the most commonly used headers, including those for caching control and content negotiation. Additionally, we examined HTTP proxying, detailing the different types and the specific use cases they are designed to address.

It must now be clear that proficiency in HTTP is indispensable for DevOps and DevSecOps professionals, as it enhances collaboration between development and security teams, streamlines the troubleshooting process, and fosters the creation of robust, secure applications.

CHAPTER 25

TLS

The Transport Layer Security (TLS) is a cryptographic protocol used to secure communications over a computer network. It encapsulates plaintext protocols such as HTTP and LDAP, providing identity authentication and a secure encrypted channel that ensures the confidentiality and integrity of transmitted data.

TLS was designed to address the security shortcomings of older plain-text protocols, including HTTP, LDAP, SMTP, POP3, and IMAP, which were developed at a time when security was not a primary concern.

In today's environment, where secure and confidential communication is essential, proficiency in TLS is a critical skill for DevOps and DevSecOps professionals.

Note This chapter contains a few examples you can run to see the explained concepts in action. The examples rely on the baseline lab we configured in Chapter 23. If you skipped the chapter, you must go back and at least set up the baseline lab.

Transport Layer Security (TLS) Protocol

The Transport Layer Security (TLS) is a cryptographic protocol designed to provide secure communication over a computer network.

It is widely used to encapsulate clear-text protocols such as `HTTP`, `LDAP`, `SMTP`, and so on, enciphering communications and ensuring integrity through message authentication codes.

In addition to that, TLS also verifies the identities of the parties involved in the communication, typically using digital certificates. The protocol evolved through several iterations (TLS 1.0, 1.1, 1.2) to the current TLS 1.3.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_25

Note OpenSSH does not use TLS or SSL protocols; it employs its own protocol for secure communication and encryption.

In Red Hat family distributions since 8 and above it is possible to control the configured cryptographic policy by using the `update-crypto-policies` command-line tool.

For example, to view the current policy, just run

```
update-crypto-policies --show
```

Available policies are as follows:

- `DEFAULT`: Accepts TLS 1.2 and 1.3, as well as the IKEv2 and SSH2 protocols. The RSA keys and Diffie-Hellman parameters are accepted if their size is at least 2048 bits.
- `FUTURE`: Like defaults, but increases the minimum size for RSA keys and Diffie-Hellman parameters to 3072 bits and does not allow the use of SHA-1 in signature algorithms.
- `LEGACY`: The same as DEFAULT, but it also accepts TLS 1.0 and 1.1 protocols and lowers the minimum size for RSA keys and Diffie-Hellman parameters to 1023 bits. It also accepts old (and so weak) algorithms such as DSA, 3DES, and RC4.
- `FIPS`: Conforms with the FIPS 140 requirements. For more information on FIPS, you can refer to the official specifications published on the NIST website (`https://csrc.nist.gov`).

Note Mind that setting the `FUTURE` cryptographic policy on Internet-facing devices may cause compatibility problems with some older clients, whereas setting the `LEGACY` policy violates several regulations. For this reason, you must always carefully think about what to do, discussing it with your security team.

Just as an example, the statement for configuring the LEGACY cryptographic policy (**don't run it!**) is

```
sudo update-crypto-policies --set LEGACY
```

Mind that changes take place only after restarting the system.

TLS Handshake

The purpose of the TLS handshake is negotiating the parameters to spin up a secure tunnel using the most secure ciphers supported on both server and client and exchange the shared key necessary to encrypt and decrypt data in a secure way.

It is a critical stage that unfortunately sometimes, especially with old or bugged clients, may not work as expected, so it is very important to understand how it works to be able to troubleshoot it when necessary.

Table 25-1 summarizes the TLS handshake process phase by phase.

Table 25-1. *TLS handshake*

Phase	Description
Client Hello	The client sends a "Client Hello" message containing a random number (client random), along with a list of supported TLS versions, cipher suites, and compression methods. It also includes extensions such as the `Server Name Indication` (SNI), which indicates the hostname the client is trying to connect to.
N.A.	The server processes the incoming `Client Hello` message by checking the TLS version and cipher suites. It selects the highest version and the most secure cipher suite that it supports from the client's list. If there is a mismatch (e.g., unsupported TLS version or cipher suite), the server sends a "handshake failure" alert. This alert does not necessarily terminate the connection: the client may choose to retry with different options.
Server Hello	The server responds with a "Server Hello" message, which includes a random number (server random), along with the chosen TLS version, cipher suite, and compression method (if applicable).

(*continued*)

Table 25-1. (*continued*)

Phase	Description
Server Certificate Exchange	The server sends its X.509 certificate to the client for validation.
N.A.	The client validates the server certificate by constructing the trust chain, checking that the certificate is not expired, and verifying that the `Common Name` (CN) or any of the `Subject Alternative Names` (SAN) match the requested Fully Qualified Domain Name (FQDN) or IP address.
Client Certificate Exchange	If mutual TLS (mTLS) authentication is required, the client sends its own certificate to the server for validation.
N.A.	The server validates the client certificate similarly by constructing the trust chain and checking for expiration. This validation is often performed using a dedicated truststore containing only the certificates of the ROOT CAs that are entitled to issue client certificates for that specific service.
Temporary Shared Key Generation	The process continues differently depending on whether the selected cipher suite supports Diffie-Hellman (DH) key exchange: • If DH is not supported, the client generates a temporary pre-master secret, encrypts it using the server's public key (from its certificate), and sends it to the server. • If DH is supported, both the client and server generate their own ephemeral key pairs and exchange the DH parameters along with the public keys. They then use the DH algorithm to derive a shared secret from the DH parameters and the received public key.
Session Keys Generation	From this point onward, the temporary shared key generated using DH or the pre-master secret generated by the client is used, along with the two random numbers (client random and server random), to generate the session keys that will be used for encrypting data during the TLS session.

Server Name Indication (SNI)

As we discussed, with HTTP virtual hosts, the server selects the appropriate virtual host for serving the request by checking the Host HTTP header. Similarly, with TLS, a mechanism is needed to select the correct certificate that has a `Common Name` (`CN`) or a `Subject Alternative Name` (`SAN`) matching the requested host. Without such a mechanism, the server won't know which certificate to send to the client, resulting in the default certificate being sent. The problem here is that if the client encounters a certificate that does not match, it considers it invalid and drops the connection.

The Server Name Indication (SNI) extension was designed to address this problem: it enables the client to specify the target host directly during the TLS handshake. This allows the server to select the appropriate certificate and present it to the client for validation.

For example, with `openssl`, it is possible to specify the SNI using the `-servername` parameter.

To better illustrate this concept in action, since Google uses the same endpoint for multiple HTTPS services, we can use it along with the `openssl` command-line utility.

For example, the following statement connects to the `google.com` endpoint, specifying that we are willing to communicate with the `google.ch` service:

```
openssl s_client -connect google.com:443 \
-servername google.ch 2> /dev/null | grep -A 11 "0 s:CN"
```

As you can see, the returned certificate chain is the one for the `google.ch` service:

```
 0 s:CN=*.google.ch
   i:C=US, O=Google Trust Services, CN=WR2
   a:PKEY: id-ecPublicKey, 256 (bit); sigalg: RSA-SHA256
   v:NotBefore: May 12 08:45:01 2025 GMT; NotAfter: Aug  4
 1 s:C=US, O=Google Trust Services, CN=WR2
   i:C=US, O=Google Trust Services LLC, CN=GTS Root R1
   a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
   v:NotBefore: Dec 13 09:00:00 2023 GMT; NotAfter: Feb 20
 2 s:C=US, O=Google Trust Services LLC, CN=GTS Root R1
   i:C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root
   a:PKEY: rsaEncryption, 4096 (bit); sigalg: RSA-SHA256
   v:NotBefore: Jun 19 00:00:42 2020 GMT; NotAfter: Jan 28
```

Note To terminate the connection, just press CTRL+C.

Now, let's retry the same statement but this time specifying we are willing to communicate with the `google.it` service:

```
openssl s_client -connect google.com:443 \
-servername google.it 2> /dev/null | grep -A 11 "0 s:CN"
```

As you should expect, the returned certificate chain this time is that of the `google.it` service.

Note SNI is supported also by other command-line tools, including `curl`, which is commonly used for testing endpoints or embedding service calls into scripts. To enable SNI when using the `curl` command-line utility, you can connect directly to the target service URL and use the `--resolve` option to specify the mapping of the target service to its IP address. For example, if the IP address of the endpoint serving multiple services, including `google.ch`, is `142.250.190.78`, you would type the following command: `curl -vsL --resolve google.ch:443:142.250.190.78` https://google.ch/.

TLS Server Certificate Validation

As discussed, during the TLS handshake, the TLS server's certificate is validated by the client. This process involves the standard Public Key Infrastructure (PKI) certificate validation, which includes constructing the certificate chain, checking that it is trusted, and verifying that the certificate is not expired.

Additionally, during the TLS handshake, the client goes beyond this by also checking that the server's certificate `Common Name` (`CN`) or any of the `Subject Alternative Name` (`SAN`) matches the requested host's Fully Qualified Domain Name (FQDN) or IP address.

To see this check in action, we can spin up a TLS endpoint using the `https.py` Python script as we did in Chapter 23.

First, change to the directory:

```
cd ~/devsecops-pillars/ch23
```

then just run

```
./https.py -c wildcard.t1-t3.carcano.corp.crt \
-k wildcard.t1-t3.carcano.corp.key
```

This script spins up a very basic HTTPS service on port 8443, enabling clients to download any file in its working directory.

Leave it running and open another terminal. Once done, as we previously did in Chapter 23, try connecting to it as follows:

```
curl --cacert \
~/devsecops-pillars/ch23/wildcard.t1-t3.carcano.corp.crt \
https://127.0.1.1:8443/v3.ext
```

this time, even if we are explicitly trusting the certificate, the connection attempt fails with the following error:

```
curl: (60) SSL: no alternative certificate subject name matches target host
name '127.0.1.1'
```

This is caused by trying to connect to the `127.0.1.1` IP address: despite this being a valid IP address bound to the loopback interface, conversely from the IP address `127.0.0.1`, as the message clearly says, the IP address `127.0.1.1` is not declared anywhere in the certificate, neither in the `Common Name` nor in the `Subject Alternative Names`.

So, since the certificate mismatches this destination, then the connection is immediately dropped by the client.

Now, let's retry running the same `curl` statement using the `dnf.t1.carcano.corp` and `www.t2.carcano.corp` FQDNs and even the `127.0.0.1` IP address: it must always run, since their FQDN match the wildcard domains claimed in the certificate's `Common Name` or in the `Subject Alternative Names` and the `127.0.0.1` IP address is listed among the `Subject Alternative Names`.

When done, quit this terminal by typing `exit` and go back to the first terminal from where we launched the `https.py` script.

Note Remind to terminate the running `https.py` instance by pressing `CTRL+C`.

TLS Server Certificate Bundle

As discussed, building and verifying a certificate chain requires access to every certificate in that chain. In typical scenarios, like HTTPS web servers, server administrators don't control the clients. Best practices dictate that clients usually only have ROOT CA certificates installed.

If a TLS server certificate is signed by an intermediate CA, the client won't be able to build the full certificate chain because it lacks the intermediate CA certificates.

Let's demonstrate this issue with a hands-on lab. We'll set up a TLS endpoint using the `https.py` Python script, as we did before.

Note This lab requires connecting to the host specified as the Common Name (CN) in the certificate you requested from Let's Encrypt in Chapter 23. You'll need to update the example to use your actual CN. Also, as in Chapter 23, add an entry for this host in your `/etc/hosts` file. Specifically, append the CN (e.g., `www.carcano.ch`) to the line starting with `127.0.0.1`.

This time we run the `https.py` Python script using the certificate we requested to Let's Encrypt (`0000_cert.pem`) and its related private key (`www.carcano.ch.key`):

```
./https.py -k www.carcano.ch.key -c 0000_cert.pem
```

Leave it running and, from another terminal, download the `v3.ext` file by running

```
curl https://www.carcano.ch:8443/v3.ext
```

As we should expect, `curl` fails with the following error:

```
curl: (60) SSL certificate problem: unable to get local issuer certificate
```

This is because, as we said, since the intermediate certificate is missing, `curl` cannot construct the certificate's chain.

Exit this terminal by typing `exit` and go back to the first terminal from where we launched the `https.py` script, terminating the script by pressing `CTRL+C`.

To address this certificate validation issue, the best practice is having the TLS server to provide all intermediate CA certificates along with its own certificate. This way, the client can construct and validate the complete certificate chain.

Most TLS server implementations offer dedicated options to load both the server certificate and the CA bundle, enabling the server to automatically provide the complete certificate chain to clients.

If, for any reason, the TLS server implementation does not have a dedicated option for loading the CA bundle, it is anyway possible to manually create a certificate bundle file by concatenating every intermediate CA's certificate to the TLS server's entity certificate into a single file.

This can be done using the `cat` command in a Unix-like shell with the `>>` redirect operator.

For example:

```
cat intermediate.crt >> server.crt
```

As we saw in Chapter 23, when requesting the CA-signed certificate, Let's Encrypt provides the `0001_chain.pem` file which already contains the TLS server entity certificate and the intermediate CA's certificate used to sign it.

We can easily verify this by running

```
openssl crl2pkcs7 -nocrl -certfile 0001_chain.pem | openssl pkcs7 -print_certs
```

The output confirms this file is a certificate bundle containing the `www.carcano.ch` TLS server entity certificate along with the `CN = E6` Let's Encrypt intermediate CA certificate.

Let's re-run the `https.py` script, but this time using the `0001_chain.pem` certificates bundle as the TLS server certificate:

```
./https.py -k www.carcano.ch.key -c 0001_chain.pem
```

As we previously did, leave it running, open another terminal, and download the `v3.ext` file from `https://www.carcano.ch:8443` by running

```
curl https://www.carcano.ch:8443/v3.ext
```

Since this time `curl` is able to construct and validate the whole certificate's chain, the contents are retrieved without problems.

Exit this terminal by typing `exit` and go back to the first terminal from where we launched the `https.py` script, and terminate the script by pressing `CTRL+C`.

Mutual TLS Authentication

As anticipated, by configuring Mutual TLS authentication, it is possible to require the client to authenticate using a trusted X.509 certificate: these are just certificates signed by a trusted Certificate Authority or self-signed certificates contained in the truststore. These certificates can either be user's identity certificates as well as system service certificates when the entity to be authenticated is a service.

To see Mutual-TLS in action, we can spin up the `https.py` Python script once again: in this lab we simulate a client connecting to it using the `wildcard.t1-t3.carcano.corp.crt` certificate to perform mutual TLS authentication.

To achieve this, we must first add the `wildcard.t1-t3.carcano.corp.crt` certificate to the `trust.pem` truststore file that will be used by the `https.py` script:

```
cat wildcard.t1-t3.carcano.corp.crt >> trust.pem
```

Note It is of course possible to add the `trust.pem` truststore as many certificates as necessary, no matter if they are CAs' certificates or self-signed entity certificates.

We can now spin up the service again as follows:

```
./https.py -k www.carcano.ch.key -c 0001_chain.pem \
-t trust.pem
```

Mind how this time we also used the `-t` parameter to specify the `trust.pem` truststore file.

Leave it running and, from another terminal, use `curl`, to download the `v3.ext` file from https://www.carcano.ch:8443:

```
curl https://www.carcano.ch:8443/v3.ext
```

As you should expect, this time `curl` gets immediately disconnected with the following error:

```
curl: (56) OpenSSL SSL_read: error:0A00045C:SSL routines::tlsv13 alert
certificate required, errno 0
```

This is correct, since now we configured the `https.py` Python script to authenticate only clients providing a trusted certificate.

Let's re-try the call, this time using the `wildcard.t1-t3.carcano.corp.crt` certificate, along with its related private key:

```
cd ~/devsecops-pillars/ch23
curl --key wildcard.t1-t3.carcano.corp.key \
--cert wildcard.t1-t3.carcano.corp.crt \
https://foo.carcano.ch:8443/v3.ext
```

Now everything works smoothly as it is supposed to.

For the sake of completeness, if we would provide an untrusted certificate – so a certificate not listed in the `trust.pem` truststore or signed by a CA whose certificate is not listed in the `trust.pem` truststore – `curl` would fail with the following error:

```
curl: (56) OpenSSL SSL_read: error:0A000418:SSL routines::tlsv1 alert
unknown ca, errno 0
```

Exit this terminal by typing `exit` and go back to the first terminal from where we launched the `https.py` script.

Note Remind to terminate the running `https.py` instance by pressing `CTRL+C`.

Summary

This chapter provided an overview of the Transport Layer Security (TLS) protocol.

We explored the TLS handshake process, highlighting the differences when the Diffie-Hellman key exchange is used versus when it is not.

We also discussed the Server Name Indication (SNI) extension and explained the certificate chain validation process, including mutual TLS (mTLS) authentication.

TLS is widely used to secure older plaintext protocols; in fact, what is commonly referred to as HTTPS is simply HTTP over TLS.

For this reason, it is essential for DevOps and DevSecOps professionals to be proficient in TLS to effectively support developers and QA teams in addressing the challenges that may arise when implementing this protocol.

CHAPTER 26

LDAP

For DevOps and DevSecOps professionals, proficiency in LDAP is indispensable: despite being relatively dated, the Lightweight Directory Access Protocol (LDAP) remains one of the most widely used authentication and authorization protocols in modern enterprise environments.

Nearly every enterprise-class application and service supports LDAP as an external authentication source, making it a critical component of identity management.

Additionally, many applications can offload configuration or security objects to an LDAP server, allowing for centralized management and easy accessibility from other LDAP-compatible applications.

Conversely from most of the chapters in this book, this chapter is purely theoretical – despite it providing a lot of examples, we won't run them as labs: the examples are very clear, and it would not be worth the effort to set up a proper lab for trying them. Nevertheless, it provides you with robust skills on the LDAP protocol and how to professionally operate it.

The Lightweight Directory Access Protocol

The Lightweight Directory Access Protocol – LDAP (`RFC 1777`) – is an application protocol used for accessing and maintaining distributed directory information services over an IP network.

In its first release, LDAP was used to provide a centralized directory of user authentication realms, user groups, address books, and network resources such as printers.

The objects stored in the directory are enforced using a schema, preventing the insertion of non-compliant objects.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_26

Note The protocol quickly evolved in LDAPv3 – RFC 2251: this brought several important improvements, such as internationalization, SASL support, referrals and chaining, improved search capabilities, and other interesting features.

Even after RFC 2251, there was still a lack of standardization, especially in the schemas: that was a major problem when dealing with LDAP implementations of different vendors. To overcome this shortcoming, they released RFC 2307 (1988), standardizing LDAP usage as a Network Information Service (NIS): this finally led to achieving compatibility of implementations from different vendors using these schemas. The RFC was then further improved by the draft RFC 2307bis.

Note Although the RFC 2307bis is still nowadays a draft, it contains important improvements – it not only provides standard schemas with new object classes and attributes enabling automount maps, sudo rules, and host-based access control: it also defined new features, such as the memberOf mechanism that automatically updates the memberOf attribute of user objects whenever a user is added to a group. It also clarified the standard, for example, better describing the way for supporting nested groups. For this reason, some major Unix vendors and Linux have adopted RFC2307bis (or part of it).

Having a customizable schema enables LDAP to be easily extended: for example, Microsoft Active Directory, which relies on Kerberos, LDAP, and proprietary RPC, extensively extends the LDAP schemas to provide their own objects, such as the Group Policy Objects.

Same way did FreeIPA – the upstream project for Red Hat Directory Server, bringing its own extensions such as the attributes ipaUser and ipaGroup.

Note LDAP in its basic form is a clear-text protocol: don't use in production unless wrapped by TLS (LDAPS or LDAP with StartTLS).

OpenLDAP Caveats

OpenLDAP is a very popular LDAP implementation, so I think it is worth mentioning here a few caveats about it.

OpenLDAP implements some features as overlays, which are modular extensions that can be added to the server's functionality. Some of them are automatically loaded, such as the `memberOf` overlay when using `RFC 2307bis` schemas. Other overlays instead are loaded on demand, such as the nested group membership overlay, that is, identified by the Object Identifier (OID) `1.3.6.1.4.1.4203.1.11.1`.

In addition to that, I want to offer a word of caution with OpenLDAP when using `RFC 2307bis` schema and doing replication to achieve high availability: OpenLDAP does not guarantee to replicate objects in the order they were created. This may lead to having a replica node to receive a group containing among its members a freshly created user before receiving that user. This can cause inconsistencies and access control issues, as the `memberOf` attribute may not be correctly populated.

LDAP Schema and Objectclasses

The schema is a critical component of an LDAP directory service, defining the structure and rules for the data stored in the directory.

The schema contains the definition of various object classes: these are objects used to define attributes, their types, and constraints on valid values. Each attribute has a specific type (e.g., string, integer) and syntax that determines how data is formatted.

Note The most basic LDAP schema is described in the `RFC 5412`: this provides the definitions for essential object classes (like `person`, `organizationalUnit` and `groupOfNames`) and attributes (such as `cn`, `sn`, and `mail`)

Object classes can be of two types: `Structural` and `Auxiliary`. Conversely from `Auxiliary` object class, `Structural` object class

- In addition to optional attributes, it can also provide mandatory ones.
- Can be subclassed, deriving other structural object classes.
- When subclassed, can have only one parent object class – so no multiple inheritance.

The only notable exception is the `top` structural object class, which is special because it cannot have any kind of attributes.

Structural object classes directly derived from the `top` object class are said to be primary.

Note Although when dealing with OpenLDAP, often documentation refers to schema files; that was a very old and now deprecated way for storing the schema. The current way is to store it in the LDAP database itself. This improvement was mostly due to easing schema replication.

Hierarchical Structure

In LDAP, the structure of the directory is defined by the Directory Information Tree (`DIT`), which organizes entries in a hierarchical manner.

For example, consider the Carcano fictional corporation using `carcano.corp` as its private domain, operating in the US and Switzerland in the Zurich and Tessin cantons.

It's `DIT` may be as follows:

```
dc=carcano,dc=corp
 |-- dc=us
 \-- dc=ch
      |-- cn=Users
      |   |-- ou=Tessin
      |   |    |-- uid=mcarcano
      |   \-- ou=Zurich
      |        \-- uid=jdoe
      \--cn=Groups
          |-- ou=Tessin
          |     |-- cn=DevOps
          |     \-- cn=Developers
          \-- ou=Zurich
                |-- cn=DevOps
                \-- cn=Developers
```

Mind that this is just an example suited to fit this specific use case, using a single authentication realm (`carcano.ch` in this case).

Another use case could be providing a multi-tier structure for different customers with a dedicated authentication realm for each one. This use case clearly requires a completely different structure.

In LDAP terms, the full path to an object within the DIT is called `Distinguished Name` (`DN`) – for example, `cn=Users,dc=ch,dc=carcano,dc=corp`.

The part of the `DN` that uniquely identifies an entry within its parent context is instead called Relative Distinguished Name (`RDN`). This corresponds to the first token of the DN.

For example, `cn=Users` is the RDN of `cn=Users,dc=ch,dc=carcano,dc=corp`.

As you can easily guess, the DN uniquely identifies an object within an LDAP directory.

In the above example, the DIT root object is `dc=carcano,dc=corp`. This kind of object, since it can contain other objects (`dc=us`, `dc=ch`), is called a container object. In this example, both `dc=us` and `dc=ch` are still container objects, since they contain `cn=Users` and `cn=Groups`, which are container objects themselves.

Designing a proper DIT structure is a critical point for running an effective directory, providing both optimal performances during lookups as well as minimizing efforts when maintaining security permissions: the LDAP standard indeed allows to assign specific permissions or even to delegate administrative rights on a per-subtree basis.

To facilitate this process, they created a few object classes that are valid container objects but with different nuances.

Table 26-1. *LDAP container objectclasses*

Objectclass	Description
container	It is just a general-purpose container object.
dcObject	It suggests it is part of a domain, and so the directory tree originating from it entirely belongs to that domain. This helps, for example, when it comes to setting up replications, having dedicated replicas for different domains.
organizationalUnit	It suggests that the directory tree originating from it is a subdivision within an organization, such as a branch, and so it is likely to have its own policies. It helps, for example, to structure branch-specific or OU-specific delegations or ACLs.

To help you to better figure out how these object classes can be used, here is the same DIT we presented before, this time represented highlighting the object classes between parentheses:

```
dc=carcano,dc=corp (dcObject)
 |-- dc=us (dcObject)
 \-- dc=ch (dcObject)
      |-- cn=Users (container)
      |   |-- ou=Tessin (organizationalUnit)
      |   |    |-- uid=mcarcano
      |   \-- ou=Zurich (organizationalUnit)
      |        \-- uid=jdoe
      \-- cn=Groups (container)
          |-- ou=Tessin (organizationalUnit)
          |     |-- cn=DevOps (groupOfNames)
          |     \-- cn=Developers (groupOfNames)
          \-- ou=Zurich (organizationalUnit)
                |-- cn=DevOps (groupOfNames)
                \-- cn=Developers (groupOfNames)
```

A structure like the above one not only provides flexibility and optimal performances, but also requires a very minimal administrative effort: by leveraging the OU containers, it is quick and easy to configure an administrative delegation affecting all the items in that subtree, thus reducing the number of necessary privilege rules.

The outcome is that every corporate branch can manage its own users and groups and even create groups with the same name with different purposes and or members.

Note In a LDAP tree the unique key is obviously the DN: a structure like the above one, although from the organizational perspective is very efficient, has the pitfall of allowing the reuse of the same uid attribute multiple times in different parts of the tree. When using the whole directory as the security realm for user authentication, this is not acceptable, since users' logins are checked against the `uid` attribute. For this reason, many LDAP server implementations provide a plugin that if enabled guarantees not to be able to create objects with the same uid even in other parts of the directory tree if one with that `uid` already exists. OpenLDAP, for example, supports this by enabling the `slapo-unique` overlay.

Base DN

The Base DN is the entry point in the directory tree from where a LDAP lookup or search operation starts.

It is crucial to specify a proper Base DN for each specific search, not only for performance reasons — since a more specific Base DN can reduce the number of entries the directory server needs to search through — but also to avoid retrieving duplicate objects that may be out of the real context of the lookup, which can generate confusion.

Note The rule of thumb is to choose a Base DN that is close enough to the items you are looking for to include all relevant entries, but not so close that you risk missing important items.

Table 26-2 shows some of the best Base DNs for lookups in the previously shown DIT by use case.

Table 26-2. *Example Base DN by use case*

Base DN	Use Case
`cn=Users,dc=ch,dc=carcano,dc=corp`	Look up every user within the Swiss corporate branch
`cn=Groups,dc=ch,dc=carcano,dc=corp`	Look up every group within the Swiss corporate branch
`ou=Tessin,cn=Users,dc=ch,dc=carcano, dc=corp`	Look up every user located in the Tessin corporate office
`ou=Zurich,cn=Groups,dc=ch,dc=carcano, dc=corp`	Look up every group used by the Zurich corporate office

Instantiating Objects

An object is defined specifying a DN followed by a list of object classes combined according to the following rules:

- The list must contain the `top` object class.

- There can be only one `primary structural` object class.
- If any subclassed `structural` object class is listed, its parent must be listed too.

In addition to that, these object classes must not have conflicting attributes, and all mandatory attributes must be set with a value.

Let's have a look and comment the definition of the root object of the `DIT` we used in the previous example:

```
dn: dc=carcano,dc=corp
objectClass: top
objectClass: domain
objectClass: dcObject
dc: carcano
```

The object class hierarchy is `top` -> `dcObject` and `top` -> `domain`.

Let's see another example, this time using the container objectclass:

```
dn: cn=Users,dc=ch,dc=carcano,dc=corp
objectClass: top
objectClass: container
cn: Users
```

Now a more complex example, like the definition of a user account:

```
dn: uid=mcarcano,ou=Tessin,cn=Users,dc=ch,dc=carcano,dc=corp
objectClass: top
objectClass: person
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: posixAccount
uid: mcarcano
cn: Marco Antonio Carcano
sn: Carcano
mail: mcarcano@carcano.corp
telephoneNumber: +41 (0)65 868 123 456
givenName: Marco Antonio
displayName: Marco Antonio Carcano
```

```
userPassword: {SSHA}hashed_password
loginShell: /bin/bash
homeDirectory: /home/mcarcano
gidNumber: 1000000
uidNumber: 1000000
```

The objectclass hierarchy is `top` -> `person` -> `inetOrgPerson` -> `organizationalPerson` -> `posixAccount`, where `posixAccount` is the only auxiliary objectclass.

The `userPassword` contains the hashed user's password that can be generated by launching a shell and running

```
slappasswd -h {SSHA} -s the_password
```

Note Advanced setups may use GSSAPI to exploit Kerberos for authentication. In such a setup, the above object has also the `krbPrincipal` object class and replaces the `userPassword` attribute with the `krbPrincipalName` attribute, having a value in the format `username@REALM`, such as `mcarcano@CARCANO.CORP`. In such a scenario, binding to the directory using the `ldap` command-line utility requires already having got the Kerberos TGT (i.e., running `kinit`) and passing the `-Y` command-line option to the `ldap` command.

LDAP Search Filters

We have now learnt enough to begin talking about LDAP search filters: these are expressions providing the criteria for selecting matching objects.

The basic structure of an LDAP search filter is

```
(attribute operator value)
```

Note Mind the enclosed parentheses – they are mandatory!

Where

- `attribute` is the name of the attribute you want to use as matching criteria (e.g., `cn`, `uid`, `mail`)
- `operator` is the comparison operator, such as `=`,`=*` and so on
- `value` is the value to match

Note It is also possible to include object classes as matching criteria by specifying the word `objectclass` as attribute and the objectclass' name to match as value. For example, the filter for matching only `posixAccount` objects is `(objectclass=posixAccount)`

The commonly used operators are summarized in Table 26-3.

Table 26-3. *Example Base DN by use case*

Operator	Meaning
=	Exact match
=*	Wildcard match (ends with)
=~	Proximity match (sometimes called approximate match). It indicates an approximate match should be performed – for example, `cn~=John` could find entries with first names like "Jon" or "Johan". Mind that the actual outcome depends on the LDAP implementation, since the approximate match filters lack standardization
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal
!	Logical NOT
&	Logical AND
|	Logical OR

For your convenience, let's see a few examples of LDAP filters.

Lookup objects with an `uidNumber` attribute greater than or equal to `3600`:

```
(uidNumber>=3600)
```

Lookup any LDAP object having the `mail` attribute set to `marco.carcano@carcano.corp`:

```
(mail=marco.carcano@carcano.corp)
```

Look up every object containing an email address belonging to the `carcano.corp` domain:

```
(mail=*@carcano.corp)
```

Look up the user `mcarcano` – please note the use of `objectclass` as matching criteria bound with the logical AND:

```
(&(uid=mcarcano)(objectclass=posixAccount))
```

Look up an object having `uid=mcarcano` OR `cn=Marco Antonio Carcano`. The object matches only if its `objectclass` is `posixAccount`. (in other words, … if it's a user):

```
(&(|(uid=mcarcano)(cn=Marco Antonio Carcano))(objectclass=posixAccount))
```

Look up every object having an email address different from `marco.carcano@carcano.corp`:

```
(!(mail=marco.carcano@carcano.corp))
```

Referrals

LDAP referrals are special entries used to inform the client on which LDAP server it can find a specific DN the current server is not authoritative on.

Referrals allow clients to follow search paths iterating multiple servers, ensuring that users can access data even if it is not stored locally. They are also very useful when dealing with changes in directory structures, such as merges or reorganizations, by allowing objects to be identified by different names across different servers.

An example referral is

```
dn: dc=us,dc=carcano,dc=corp
```

```
objectClass: top
objectclass: referral
dc: us
ref: ldap://ldap.us.carcano.corp/dc=us,dc=carcano,dc=corp
```

When a client encounters a referral (if it is configured to chase them), it automatically follows it, connecting to the referred server and re-submitting the original request to the new server. This mechanism is called chasing referrals.

Note Referral chasing may be enabled or disabled by default – for example, the `ldapsearch` command-line utility has chasing referrals already on, but in several libraries, including the most used ones with Python, referrals are off by default.

Bind DN

The Bind DN is the distinguished name of the user account that you want to use for authenticating to the LDAP server.

While many recent LDAP installations allow connections by specifying a username and password, the traditional method of LDAP authentication involves providing a `Bind DN` and its associated password.

For example, the Bind DN of the user having uid `mcarcano` is `uid=mcarcano,ou=Tessin,cn=Users,dc=ch,dc=carcano,dc=corp`

The ldapsearch Command-Line Client

OpenLDAP provides a convenient command-line tool that can be used to perform LDAP lookups from the shell.

On the Red Hat family distros, it is provided by the `openldap-clients` RPM package. You can install it by running

```
sudo dnf install -y openldap-clients
```

Note As we said LDAP at its basic is a plain text protocol, so by default it connects to the standard LDAP port (tcp/cov89) and uses unencrypted communications. The recommended way is instead to start TLS immediately after the connection (LDAP with StartTLS) or to directly use LDAP over SSL (LDAPS) and connect to port `tcp/636`.

You can ask the LDAP server to dump the whole DIT by providing the `-LLL` option. This is a fictional example of `ldapsearch` statement:

```
ldapsearch -H ldap://ldap.ch.carcano.corp -D \
"uid=admin,ou=Tessin,cn=Users,dc=ch,dc=carcano,dc=corp" -W \
-b "dc=carcano,dc=corp" -LLL
```

The options used are as follows:

- `-H` is the LDAP's URL – mind the `ldap://` tells the client to connect to port `tcp/389`.
- `-D` is the Bind DN.
- `-W` asks the tool to prompt for interactively entering the password.
- `-b` is the Base DN.

As we said, the best practice is to use an encrypted connection (LDAP with StartTLS or LDAPS).

They reworked the statement for running the same fictional lookup, this time using LDAP with start TLS (which is enabled by the `-Z` option), is

```
ldapsearch -H ldap://ldap.ch.carcano.corp -Z -D \
"uid=admin,ou=Tessin,cn=Users,dc=ch,dc=carcano,dc=corp" -W \
-b "dc=carcano,dc=corp" -LLL
```

If everything is properly set, you must get the same outcome as before, listing all the contents the bind DN user you specified has access to.

If instead you get an error message like:

```
certificate verify failed (self-signed certificate in certificate chain)
```

It means the LDAP's server certificate is signed by a CA not listed in your system truststore. In such a scenario, you must first download the LDAP service's certificate chain into a file as follows:

```
echo | openssl s_client -connect ldap.ch.carcano.corp:636 -showcerts 2>/dev/
null | awk '/-----BEGIN CERTIFICATE-----/,/-----END CERTIFICATE-----/' >
~/ldap.ch.carcano.corp-CA.crt
```

The `~/ldap.ch.carcano.corp-CA.crt` file contains both the server certificate and the CA's certificate used to sign it. As you now must know, the best practice is to wipe the server's certificate and keep only the intermediate CA ones.

A quick way for identifying the certificates in that file is to run

```
openssl crl2pkcs7 -nocrl -certfile ~/ldap.ch.carcano.corp-CA.crt | openssl
pkcs7 -print_certs
```

The certificate containing the LDAP server's FQDN in the CN is the one to wipe – most of the time – the first one: just remove it from the `~/ldap.ch.carcano.corp-CA.crt` file.

Now, depending on your specific use case, you have three distinct choices:

- Add it to the system-wide truststore.
- Configure the `TLS_CACERT` option in the system-wide LDAP client configuration file `/etc/openldap/ldap.conf`.
- Add the `-o TLS_CACERT=~/ldap.ch.carcano.corp-CA.crt` option to the `ldapsearch` command line.

You must pick the one that best fits your real scenario: for example, if you are configuring LDAP as the system-wide authentication source for logging in to a server, you probably opt for the first one. If instead you are configuring an application linked to the LDAP library, you may opt for the second or the third.

The last connection type is the one using LDAPS – in this case the fictional statement becomes as follows:

```
ldapsearch -H ldaps://ldap.ch.carcano.corp -D \
"uid=admin,ou=Tessin,cn=Users,dc=ch,dc=carcano,dc=corp" -W \
-b "dc=carcano,dc=corp" -LLL
```

As you can see this time we specified `ldaps://` to tell the client to connect to port `tcp/636`. Of course, the same pitfall about certificate's trust we just saw applies also here.

Some LDAP services, such as Microsoft Active Directory, allow you to specify the username instead of the Bind DN. In the following example we are simulating a connection using the `administrator` user from the `carcano` realm:

```
ldapsearch -x -H ldaps://msad.ch.carcano.corp:636 -D \
"carcano\administrator" -W \
-b "cn=Users,dc-ch,dc=carcano,dc=corp" \
"(&(samAccountName=mcarcano)(objectClass=posixAccount))"
```

As you can see, in this example, I also caught the opportunity to show how to specify the `(&(samAccountName=mcarcano)(objectClass=posixAccount))` LDAP search filter.

If the server supports Kerberos authentication, after getting a valid TGT by running `kinit`, you may be able to run `ldapsearch` avoiding specifying credentials by using `GSSAPI` authentication:

```
ldapsearch -H ldaps://ldap.ch.carcano.corp:636 -Y GSSAPI \
-b "ou=Tessin,cn=Users,dc=ch,dc=carcano,dc=corp" \
"(&(uid=mcarcano)(objectClass=posixAccount))"
```

Note Kerberized LDAP authentication is highly secure, but its requirements depend on the method used. A true Kerberos SSO flow (GSSAPI) requires the client to be registered in the Kerberos realm, a properly configured krb5.conf, and the Cyrus SASL GSSAPI plugin installed. In addition to that, if you are using SASL Passthrough (LDAP server accepts a plaintext password and verifies it against the Kerberos database on the user's behalf), you also need the saslauthd, which acts on behalf of you, talking with the Kerberos server.

Other LDAP Command-Line Tools

The `ldapsearch` command-line utility is just one of several tools available for interacting with LDAP directories.

For instance, the `ldapmodify` command enables you to modify existing objects in the LDAP directory and can also be used to add new objects, either interactively or by loading them from a file.

Note For the sake of completeness, there also exists the `ldapadd` command, which in some implementations is often a symlink to the `ldapmodify` command-line tool. However, `ldapadd` is the preferred command for adding new entries, as its semantic is specifically designed for this purpose. Conversely, while `ldapmodify` can also be used to add new objects, it requires a specific input format to do so.

When it comes to deleting entries from the directory, you must instead use the `ldapdelete` command-line tool.

Since understanding the full range of LDAP command-line tools is essential for efficient directory management and operational effectiveness, I strongly recommend you to read https://tldp.org/HOWTO/LDAP-HOWTO/utilities.html and consult the related man pages (`man ldapsearch`, `man ldapmodify` and `man ldapdelete`).

Summary

This chapter equipped you with a comprehensive overview of LDAP: we didn't limit to the purely operational concepts, explaining some OpenLDAP-specific caveats, discussing an example directory structure, and delving into protocol details that should make you capable of using it in nearly every use case you may have as a professional.

For DevOps and DevSecOps professionals, understanding the LDAP protocol and its operations, such as using `ldapsearch`, is crucial for effective management of user authentication and directory services. By knowing how to operate LDAP, professionals can efficiently query and manage user accounts, groups, and other directory objects, ensuring that access rights are correctly assigned and maintained.

Moreover, familiarity with the types of objects found in an LDAP directory, such as users, groups, and organizational units, allows professionals to navigate the directory structure effectively.

This understanding is vital for troubleshooting authentication issues, implementing security policies, and integrating various applications with directory services.

A well-structured directory enhances operational efficiency and security posture, as it enables seamless user management and access control. Therefore, proficiency in LDAP not only aids in operational tasks but also plays a significant role in the overall security framework, making it a critical skill for professionals.

PART IX

Working with Containers

A must-have for DevOps and DevSecOps professionals are certainly skills in containers: running applications within containers nowadays is often the most effective way to operate, providing a very easy and convenient way for packaging and delivering the application, running them in a workspace isolated from the rest of the system.

By packaging them inside containers, it is even possible to run applications that were packaged for a specific Linux distribution on a host with a completely different one. Moreover, containers run within a confined environment that involves not only filesystem resources, but also flipping uid and gid numbers into dedicated ranges outside the system's scope, restricting the reachable processes, the IPC, and even the networking. This is a huge security improvement, compared to the legacy ways for confining applications, for example, within a chroot jail or using Sun Solaris zones.

The chapters in this section delve into this amazing quite recent technology, spanning from container management suites such as `Docker` and `Podman` to `Kubernetes`, `Rancher`, and `Helm`.

CHAPTER 27

The Containers Ecosystem

Containers offer an efficient and convenient method to isolate running processes and their runtime environments from the underlying host system. Their adoption has surged dramatically in recent years, making containerization skills not only essential for DevOps and DevSecOps professionals, but also highly valuable for developers, system engineers, QA specialists, and IT security experts across the technical IT landscape.

Since as DevOps and DevSecOps professionals, we are supposed to have a thorough understanding of the whole container ecosystem, this chapter equips you with the necessary theory, helping you familiarize with the concepts and elements that build this ecosystem.

Containers

When first encountering containers, it's essential to clarify that, despite their isolation capabilities, they are not a form of virtualization.

Unlike virtual machines (VMs), which run their own operating systems and kernels, containers share the host's kernel. Because of this fundamental difference, you cannot run Linux containers and Microsoft Windows containers on the same host, since they require a different kernel. The only way to do that is by using a virtualization layer, so using virtual machines, running the containers inside them. It is instead possible to run on the same host containers derived from different Linux distributions, as they all utilize the Linux kernel.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_27

Containers leverage features provided by the Linux kernel:

- **Namespaces**: They provide isolation for various resources, such as processes, network interfaces, and user IDs. For example, the process namespace allows a container to have its own subtree of the process tree, meaning it can only see and interact with its own processes.
- **Cgroups** (**control groups**): They are used by the kernel to manage and limit resource usage for containers. They can throttle CPU, memory, network, and I/O usage, ensuring that no single container can monopolize the host's resources.
- **Seccomp BPF**: It is a framework that allows for fine-grained control over system calls, enhancing security by specifying which syscall a container can make and which arguments are allowed.

In addition to these features, containers often utilize virtualized private networks (referred to as container networks) that are only accessible on the host. A host can create multiple container networks to ensure proper network segregation, enabling containers to share resources within the same virtualized private network as needed. These private networks can be interconnected between different hosts using overlay networks like `VxLAN` or `Geneve`, managed by software such as `Open VSwitch` and `OVN`.

Note These networking technologies belong to the so-called Software Defined Networks (SDN).

In their default setup, under the hood, container networks are implemented by creating a bridge connected to the host's network interfaces. A dedicated IP pool is reserved for assigning IP addresses to containers attached to that specific bridge/container network, and NAT is configured to allow external access from the containers.

Note Since by default the containers' traffic go through a NAT, other services in the networking environment outside the container host will see traffic generated from them as originating from the host's IP addresses.

In their default behavior, when a container is started, the host creates a virtual network device in the container's network namespace and binds it to the specified (or to the default) container network, assigning it an IP address from the pool. An `iptables` rule is then configured to allow network traffic from that IP to flow to and from the container.

Containers are spun up from container images, which provide a filesystem with all the contents necessary to run the container. Since this filesystem is read-only, when starting the container, the system adds an ephemeral read-write layer on top of it, enabling the container to save files lasting as long as the container's lifetime.

If necessary, the host can also bind-mount paths to the container's root filesystem, providing paths with access to persistent storage that can be used for saving data files or injecting configuration files.

Container Images

Container images are a convenient way to package a containerized application along with all its required dependencies, including the runtime environment. A container image can include a minimal Linux distribution with the necessary packages to run the application.

Container images utilize union filesystems like OverlayFS to manage multiple layers, where each layer is implemented as a `tar` archive containing filesystem changes.

The union of these layers provides the final layout of the container image's filesystem, with the contents of the outer layers overriding the others.

Note The contents of these layers are immutable, even the topmost layer ones: this means that once the creation of the container image is completed, you cannot add contents even to its top layer anymore. Adding content just results in creating a new outer layer.

During the build process, container images are created deriving them from existing ones, copying the layers, and adding new ones as necessary: the source image is specified using the `FROM` directive.

If it is necessary to create an image without deriving it from an existing one, just specify the `scratch` keyword as the source image: this causes the process to create the image from scratch, resulting in an image containing only the layers generated during the creation process.

Note An example of "from scratch" container image use case is packaging statically linked applications into containers. This practice enables the creation of ultra-thin container images particularly suitable when dealing with an environment oriented to micro-services.

A container image also contains metadata stored in a JSON file, which describes the layers and their configuration.

At container image creation time, each statement generates a new `tar` archive where to store contents changes, such as installed files or modified contents, committing changes at statement completion. For this reason, to keep the number of layers at a bare minimum, the best practice is to put as many SHELL statements as possible within the same pipeline.

This has pitfalls: for example, during this process, any shell statement producing changes, even just a change in the metadata of a file, such as updating the last accessed time, causes that file to be added to the outer `tar` archive. Let's illustrate this with an example.

Imagine you have a directory tree whose contents consistently have the same owner. While creating the container, you have a statement which puts some files owned by another user inside this directory tree. To restore consistency, you then have a statement with recursive `chown` on the directory to fix things: this last statement causes the change of the time attribute of every file to be updated, not only of the ones you want to fix the ownership, resulting in having all of them copied into the new layer. So, in such a scenario, the correct way is to programmatically change the ownership of only the files you just added.

Note The best practice is to keep the layer's size as thin as possible: this, besides reducing the wait time when pulling the container image from the registry, speeds up the generation of the overall union-fs layout. In addition to that, you must also keep the number of layers at its bare minimum: we will see how to do this when explaining how to create container images.

Container images are identified by a name consisting of a `label` and a `tag` separated by the colon character (`:`).

For example:

- `nginx:1.27.4`
- `nginx:1.27.4-alpine`
- `nginx:latest`

As you see from the examples, the `label` is used to better qualify the image's contents, such as the version (`1.27.4`) and the variant (`-alpine`) or just quickly identify the most current one (`latest`).

When dealing with images pulled from a registry, or intended to be pushed to a registry, the name is incorporated in a full URL including the container image registry's URI and the repository name within the container image registry.

For example:

```
docker.io/library/nginx:latest
```

where

- `docker.io` is the container image registry endpoint
- `library` is the repository name
- `nginx:latest` is the image name

Note When tagging images to be pushed to container image registries, don't put hardware architecture information in the tag: this information is automatically added as metadata to the container image. If, when pulling, an image is available for more than an architecture, the one matching the host's architecture gets automatically selected.

Container Image Registries

Just as application packages are deployed from software repositories published by repository managers, container images are deployed from repositories published by container image registries.

These registries provide a specific API for locating, pulling, and pushing container images.

By default, the service endpoint listens on port `TCP/5000` and communicates over HTTP or HTTPS.

Note Container image's initial API version was quickly replaced by `v2`: besides enhancing performances by supporting the push and pull of individual image layers and improving authentication mechanisms, the intent was also preventing information leaks from search functionalities. Unfortunately, this last was a breaking change which required users to adapt their workflows, as the `docker search` command became less effective, necessitating the use of dedicated tools like Skopeo.

Public Online Container Registries

The most popular container image registry is **Docker Hub** (`https://hub.docker.com/`).

In addition to a command-line API, it offers a convenient web UI for searching container images and listing available tags.

Docker Hub hosts official images from various vendors and is widely used by contributors to share their projects. Note that non-individual usage of Docker Hub is subject to subscription plans.

Note Official images are available beneath the path `https://docs.docker.com/docker-hub/official_images/`.

Docker Hub provides just a way for publishing container images, leaving the user the effort to implement security on his own.

Another popular online container registry is **Quai.io**, which goes beyond these basic features, providing security features such as built-in security scanning and granular user management.

Although **Quai.io** has a smaller marketplace compared to Docker Hub, it can be an interesting choice for organizations keen on delivering their products without having to implement security checks and access policies by themselves.

Other notable public registries include **Google Container Registry** (**GCR**) and **Amazon Elastic Container Registry** (**ECR**) or **Huawei's Software Repository for Container** (**SWR**), which provide integrated solutions for their respective cloud platforms.

Private Online Container Registries

Many organizations prefer to maintain their own private container registries to control access to proprietary images and ensure compliance with security policies. Examples include Red Hat's registry (`registry.redhat.io`) and Oracle's container registry (`container-registry.oracle.com`).

On-Premises Container Registries

For organizations heavily leveraging containers, implementing an on-premises container registry is a best practice.

In addition to that, mind also that some regulations forbid storing in-house developed software (and so the related container images) outside an organization.

On-premises container registries serve as both a local source to publish custom images and a local cache for publicly available ones. This improves delivery speed and resilience against network outages.

The simplest way to run on-premises container registries is using the Docker Registry v2 container image (`registry:2`), which provides just the registry service developed by Docker.

This solution has the pros of a very quick and easy setup, since it requires just to create the container, configure it by setting a few environment variables and bind-mounting the host paths to make its contents persistent.

However, this basic setup lacks essential security features like role-based access control (RBAC) and image vulnerability scanning.

For a more complete solution, consider using products such as **Harbor**, an open-source container registry that extends Docker Registry v2 with features like RBAC, image vulnerability scanning, and replication.

Other viable options are using multi-purpose repositories, such as **Sonatype's Nexus** or **JFrog's Artifactory**, or container registries bundled with SCM implementations combined with CI, such as **Gitea** or **Gitlab**.

Summary

This chapter is only an introduction to the container world, providing just an overview of the concepts that are necessary to know to work with them.

We have learned basic concepts such as what containers and container images are, as well as how they are published through online or local container image registries.

This information is not just a required pillar before moving to the next chapter of this section but also provides the theoretical concepts to understand the underlying technology of containers.

CHAPTER 28

Docker and Podman

Docker and Podman stand out as the leading container management tools in modern DevOps and cloud-native environments. Both suites enable seamless container management, allowing users to spin up containers effortlessly, build custom container images, and manage container registries by pushing or pulling images as needed.

For DevSecOps professionals, proficiency in Docker and Podman is essential. Mastering these tools ensures they can effectively support development teams with containerization workflows and assist QA teams in testing environments. Their expertise helps troubleshoot container-related issues, optimize build pipelines, and enforce security best practices across the software development lifecycle.

This chapter delves into both container management suites, providing several lab and real-life scenarios requiring you to run statements to better understand the topics and see the explained concepts in action.

Note Before going any further, change to this **chapter's workspace directory** by running `cd ~/devsecops-pillars/ch28`.

To avoid being distracted by setting up the operating environment before each specific example, we set up a baseline lab right now.

A few practical examples require pulling container images from the `reg.t1.carcano.corp` container image registry we will install on the lab VM.

Since the examples are supposed to be executed on the same machine running the registry, also to avoid invasive changes to the networking environment, we configure hostname resolution using the `/etc/hosts` file.

Just add a line like the following one:

```
192.168.66.177 reg.t1.carcano.corp
```

where `192.168.66.177` is the lab's host IP address.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_28

Note To find the IP address of your lab's virtual machine (VM), run the command `ip -4 a` in the VM's terminal. This will list the IPv4 addresses associated with its network interfaces. If you launched the VM using the `Vagrantfile` provided with the book, the relevant IP address is typically assigned to the `eth1` interface. If you're unsure which IP to use, check for the one that shares the same subnet as your host machine's IP address. You can verify this by comparing network prefixes (in my own scenario, both starting with `192.168.66.x`).

Container Management Suites

As explained in the previous chapter, deploying containers involves a series of manual steps that, while based on standard facilities and established software, can be time-consuming and error-prone. While it is possible to manage these steps manually, doing so can lead to inconsistencies and inefficiencies in the deployment process.

Recognizing this challenge, **Docker Inc**. was the first company which simplified container management. They created a suite of tools collectively known as **Docker**, which automates many of the steps involved in container deployment and management. This resulted not only in streamlining the process but also in reducing the likelihood of human error, making it easier for developers and operations teams to work with containers. As a result, the term "Docker" became synonymous with container technology for a considerable time.

However, despite the advancements brought by Docker, the platform had its limitations. One of the most significant drawbacks was not adhering to an open standard, which created challenges for interoperability and compatibility across different container solutions.

To address this issue, the **Open Container Initiative** (**OCI**) was established in June 2015. The OCI aims to create open standards for container image formats and runtime specifications, ensuring that containers can be used consistently across various platforms and tools.

Today, it is important to distinguish between the terms Docker and containers. The term "container" now refers to any technology that complies with the standards set forth by the OCI, rather than being exclusively associated with Docker.

This shift has allowed for the emergence of other container-related projects and tools, broadening the ecosystem.

Docker

Docker was indeed one of the first successful implementations of container technology, building on earlier efforts such as LXC (Linux Containers).

While LXC provided a viable way to create and manage containers, it did not gain widespread popularity, probably because of its complexity and the lack of user-friendly tools for operation and management.

Docker was able to revolutionize the container creation process by simplifying it and fitting everything into three main components:

- **Docker CLI**: The command-line interface (CLI) provides a straightforward way to execute commands for building, running, and managing containerized applications.
- **Docker Daemon** (`dockerd`): The background service that manages Docker containers. It listens for API requests via a REST API endpoint or a UNIX socket and is responsible for handling container lifecycle operations, such as starting, stopping, and removing containers.
- **Containerd**: An industry-standard core container runtime used by Docker to manage the container lifecycle. It handles the low-level operations of running containers, including image transfer and storage, container execution, and supervision. To achieve this, it relies on lower-level tools like `runc`, which is responsible for creating and running containers according to the Open Container Initiative (OCI) specifications.

Despite Docker's success in making container management more accessible, its initial design raised several concerns, particularly regarding security.

The most criticized one is that the Docker daemon runs as the `root` user by default, posing the risk that any container running under Docker could potentially gain elevated privileges on the host system.

If you want to try Docker and you are running a modern Red Hat Linux family distribution, you must first make sure Podman and Buildah are not installed. You can easily check this by running

```
rpm -q podman buildah
```

If any of them is installed, just remove it. For example, to remove both, run

```
sudo dnf remove -y podman buildah
```

Since Red Hat's official container management suite is Podman, Red Hat does not provide Docker's RPM packages anymore.

That means you must add the official Docker DNF repository:

```
sudo dnf config-manager --add-repo \
https://download.docker.com/linux/rhel/docker-ce.repo
```

You can now safely install Docker by running

```
sudo dnf install -y docker-ce docker-ce-cli \
containerd.io docker-buildx-plugin docker-compose-plugin
```

Lastly, start the `docker` service and enable it to start at boot:

```
sudo systemctl enable --now docker
```

Once installed, you must add every single user you want to be able to run `docker` statements to the `docker` group.

For example, you can add the current user as follows:

```
sudo usermod -aG docker $(whoami)
```

Once done, log off and log on again to have the group change applied to the session.

Podman

Podman is a Red Hat project that was first released in 2018. It was designed to provide a daemonless container management experience and to comply with the Open Container Initiative (OCI) standards for container images and runtimes.

This means that Podman does not require a long-running background service to manage containers, allowing users to run containers directly from the command line and interact with lightweight processes managed by `runc`.

In addition to the Podman core functionality implemented by the `podman` command-line utility, the Podman project includes a set of complementary container management tools, including `skopeo` for image management and `buildah` for building container images.

Podman was designed not only to have the `podman` command-line interface that closely resembles Docker's but also to include a RESTful API that is compatible with Docker's API. This compatibility allows tools and applications that interact with Docker to work seamlessly with Podman as well. As a result, switching from Docker to Podman typically requires minimal effort, with little to no rework needed for existing Docker-based workflows.

It is worth mentioning that, unlike Docker, which by default stores container images under `/var/lib/containers`, Podman stores them in the same path used by Docker only when the user running Podman is `root`. For non-root users, Podman stores container images and data under `~/.local/share/containers`. This design choice enhances security and allows users to manage their containers without requiring elevated privileges.

Other notable features of Podman include

- **Kubernetes YAML generation**: Podman can easily generate Kubernetes pod YAML manifests by running the `podman generate kube` command. This feature simplifies the process of transitioning from local development to Kubernetes deployment.
- **Systemd integration**: Podman can generate Systemd unit files to start containers, providing an operational experience like managing traditional services. Additionally, Podman containers can be directly managed by the Systemd process, allowing for better integration with the host's service management.

For the sake of completeness, **but don't do it now**, if you want to try Podman, you can install it by running

```
sudo dnf install -y podman podman-compose
```

Note If you installed Docker, remember to remove it before installing Podman. As a general example, run `sudo dnf remove -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin`.

Operating Container and Images

Having learned the necessary theory, we are now ready to have a look at how to operate in a containerized environment.

To avoid redundancy, and since it is the most popular container management suite, the following examples are based on Docker, but mind they mostly work with Podman as well by simply replacing the word `docker` with `podman`.

Note Both Docker and Podman enable users to override the global configuration file using user specific ones. Docker overrides can be put in the `${HOME}/.docker/config.json` file, whereas Podman in the `~/.config/containers/registries.conf` file.

To avoid redundancy while explaining, the examples in this chapter always refer to the global configuration file.

Operating with Container Image Registries

The very first operations to learn are pulling images from and pushing them to container image registries.

Local Container Image Registries

As explained in the previous chapter, in addition to using online container image registries, you can of course run your own registries. The advantage of this is avoiding continuously pulling from the online registries as well as having a place for hosting the repositories for your custom container images.

When dealing with local registries, the best practice is to always require authentication to use them, no matter if you are pushing or pulling, and of course, to guarantee confidentiality, configure them to use TLS.

Spin Up a Local Container Registry

This chapter's workspace directory contains a few sample files you can use to spin up a very simple but effective Docker registry.

The example consists of an NGINx container used as a reverse HTTP proxy to the Docker registry container.

Using a reverse proxy enables us to terminate the TLS connection and manage user authentication in a more effective way. The sample configuration uses a local authentication realm, but you can easily rework it to implement LDAP authentication.

Note This is just an example to enable you to play with container image registries: if you want a full-featured product, consider using Harbor.

To spin it up, beneath `/opt/registry`, create the `conf` directory to inject configuration files and the `data` directory to provide persistency:

```
REG_PATH=/opt/registry
sudo mkdir -m 755 -p {REG_PATH} ${REG_PATH}/conf \
${REG_PATH}/data
```

Since the resources to be used are in the `registry` subdirectory of this chapter's workspace, change to it:

```
cd ~/devsecops-pillars/ch28/registry
```

then, copy the example NGINx container's configuration:

```
sudo cp nginx.conf ${REG_PATH}/conf
```

Note For better understanding, examine the contents of the `nginx.conf` file.

To implement TLS, copy the wildcard certificate and key we generated in Chapter 23:

```
sudo cp ~/devsecops-pillars/ch23/wildcard*.key \
 ${REG_PATH}/conf/server.key
sudo cp ~/devsecops-pillars/ch23/wildcard*.crt \
 ${REG_PATH}/conf/server.crt
```

We then need to generate a username and password pair for authenticating to the registry.

In this example we are spinning up a temporary `httpd:2` container, using it to run the `htpasswd` program for creating the `ci_user` user with `grimoire` as password, adding them to the `${REG_PATH}/conf/htpasswd` file:

```
docker run --rm --entrypoint htpasswd \
docker.io/library/httpd:2 -Bbn ci_user grimoire \
| sudo tee -a ${REG_PATH}/conf/htpasswd
```

Note If you are using Podman, replace the above `docker run` with `podman run`. You can repeat this step as many times as you want to create additional users.

Finally, let's spin everything up by simply running

```
docker compose up -d
```

Note If you are using Podman, run `sudo podman compose up -d`. Since Podman deployments are user specific, mind also to always prepend `sudo` to command statements related to this deployment. For example: `sudo podman ps -a`, `sudo podman start nginx docker-registry` and so on.

If you are new to containers and could not make sense of everything explained so far, don't worry: we delve into all of this in the rest of the chapter.

Authenticating to Registries

As we said, conversely from public online container image registries, the private ones most of the time require authentication even just for pulling container images.

To avoid repeatedly asking for credentials, both Docker and Podman authenticate container image registries using cached credentials, asking to enter them only the first time.

Docker by default stores the base64-encoded credentials in the `${HOME}/.docker/config.json` file, whereas Podman stores them in the file `${XDG_RUNTIME_DIR}/containers/auth.json`.

The credential cache file contains dedicated entries with the credentials for each accessed container image registry.

Credentials are added by running the `docker login` statement followed by the container image registry's URI.

For example:

```
docker login reg.t1.carcano.corp
```

When asked for the username, enter `ci_user` and `grimoire` as the password.

Note If you are using Podman, replace `docker login` with `podman login`.

In this case you will get the following error:

```
Error response from daemon: Get "https://reg.t1.carcano.corp/v2/": tls:
failed to verify certificate: x509: certificate signed by unknown authority
```

This is expected, since we configured the container image registry with a certificate not signed by a globally available CA.

We will see how to fix this in the next paragraph, which shows how to trust registries with TLS certificates not signed by globally available CAs.

For the sake of completeness, mind that we must remove cached credentials as soon as they are no longer necessary (e.g., when we are sure we don't need to push or pull for some time).

This is performed by running the `docker logout` statement followed by the container image registry's URI. For example:

```
docker logout reg.t1.carcano.corp
```

If you are working with Podman, just replace `docker logout` with `podman logout`.

Note Since cached credentials are not encrypted, the best practice is to remove them as soon as they are no longer necessary. In other words, during sessions, as soon as it is no longer necessary to interact with a specific container image registry, the best practice is to immediately log out from it.

In environments with security concerns, a more secure approach is to set up an external credential store by configuring Docker to use a credential helper program.

The currently available helpers are

- D-Bus Secret Service: https://github.com/docker/docker-credential-helpers/releases
- Pass: https://github.com/docker/docker-credential-helpers/releases
- Apple macOS keychain: https://github.com/docker/docker-credential-helpers/releases
- Microsoft Windows Credential Manager: https://github.com/docker/docker-credential-helpers/releases

Trusting Registries and CA Certificates

Unless they are published using a public domain, local container image registries are forced to use TLS certificates signed by private Certificate Authorities or, in the worst-case scenario, self-signed certificates.

The consequence is that to use them, you must configure Docker or Podman to trust that CA certificate or the self-signed certificate used by the registry.

The trusting process consists of creating a directory with the name of the registry's FQDN, putting it beneath `/etc/docker/certs.d` if using Docker or beneath `/etc/containers/certs.d` if using Podman.

As an example, let's fix the error we just had when running `docker login` by trusting the `reg.t1.carcano.corp` registry TLS certificate.

If you are working with Docker

```
sudo mkdir -p /etc/docker/certs.d/reg.t1.carcano.corp
sudo cp ${REG_PATH}/conf/server.crt \
/etc/docker/certs.d/reg.t1.carcano.corp/trust.crt
```

If you are dealing with Podman

```
sudo mkdir -p /etc/containers/certs.d/reg.t1.carcano.corp
sudo cp ${REG_PATH}/conf/server.crt /etc/containers/certs.d/reg.t1.carcano.
corp/trust.crt
```

You can now repeat the docker login which previously failed:

```
docker login reg.t1.carcano.corp
```

When asked for the username, enter `ci_user` and `grimoire` as the password. This time it must just work smoothly.

If you are working with Podman, just replace `docker login` with `podman login`.

Untrusted Registries

Special care must be taken when dealing with the so-called untrusted registries. In a normal registry interaction process, Docker and Podman check the security of the involved parties and objects, for example, checking the TLS certificate or validating container image signatures.

However, in some rare circumstances, you may need to disable these checks.

Note By default, Docker considers connections to untrusted registries if they resolve to the loopback IP address (`127.0.0.1`). In contrast, Podman always requires explicit configuration for untrusted registries.

In such a need, if using Docker, simply add the FQDN of the registry you want to mark as untrusted to the `insecure-registries` list in the `/etc/docker/daemon.json` file.

For example:

```
{
  "insecure-registries": ["insecure-reg.t1.carcano.corp"]
}
```

If instead you are using Podman, you must add the container image registry to the `/etc/containers/registries.conf` file, marking it as untrusted, as shown by the following snippet:

```
[[registry]]
location="insecure-reg.t1.carcano.corp"
insecure=true
```

Unqualified Search Registries

When pulling or pushing container images, it is indeed necessary to specify their fully qualified image URI, which includes the fully qualified domain name (FQDN) of the container image registry hosting the repository that provides the image.

This requirement can be limiting, as it necessitates knowledge of the specific registry where a particular image is stored. Additionally, it complicates scenarios where multiple registries host the same images for redundancy purposes.

To address this limitation, Podman offers the capability to configure unqualified search registries. These registries are consulted when pulling unqualified container images (images that do not include the registry's FQDN).

Once a suitable registry containing the requested image is identified, Podman will pull the image from that registry.

By default, Podman comes with a preconfigured list of unqualified registries, such as `docker.io` and `registry.redhat.io`.

If you need it, you can of course add your own registries to the `unqualified-search-registries` list in the `/etc/containers/registries.conf` file.

For example:

```
unqualified-search-registries = [ "reg.t1.carcano.corp", "registry.access.
redhat.com", "registry.redhat.io", "docker.io" ]
```

By default, when pulling an unqualified image, Podman will ask you to interactively choose the container image registry to use.

If you want to avoid this interactive process, change the value of `short-name-mode` to `"disabled"`.

Searching Container Images in Registries

As we discussed, the `docker search` command relies on the deprecated `v1` API, which is no longer widely used.

This functionality has been effectively replaced by the `skopeo` command-line tool: `skopeo` indeed, while being designed for managing OCI images and repositories, is also compatible with Docker `v2` images.

Since it is a distinct tool from `podman`, we must install it by running

```
sudo dnf install -y skopeo
```

Besides the default local OCI-layout directories, skopeo can interact with various API v2 registries with or without authentication. Supported registries include Docker registries, the Atomic registry, and private registries.

Table 28-1 summarizes the path schemes supported by skopeo.

Table 28-1. *Skopeo supported path schemes*

Objectclass	Description
oci:path:tag	The local OCI (Open Container Initiative) layout, where the default storage location for container images is typically beneath /var/lib. This directory serves as a shared local container image store accessible to all users on the system.
containers-storage:docker-reference	The local container image store used by Podman, which is located beneath the user's home directory. This store contains only the container images utilized by that specific user, ensuring user-specific isolation.
docker://docker-reference	Images in a Docker registry
dir:path	Local directory where container images can be retrieved or stored.
oci-archive:path[:docker-reference]	Local file containing the container image, typically in the form of a tar archive.

The available actions with skopeo include

- copy: From/to remote registry, but even between two different remote registries
- delete: Even from remote registry
- inspect: Shows properties including layers of images

Note While working with container image registries requiring authentication, skopeo can often use the credentials already set by podman login or docker login.

Here are a few skopeo operational examples which illustrate all the above path schemes:

- List all the available tags of the nginx container image in the library repository on the docker.io container image registry:

  ```
  skopeo list-tags docker://docker.io/library/nginx
  ```

- Display information of the nginx:1.27.4 container image in the library repository on the docker.io container image registry:

  ```
  skopeo inspect docker://docker.io/library/nginx:1.27.4
  ```

- Display detailed information of the nginx:1.27.4 container image in the library repository on the docker.io container image registry, including how each single container image layer was generated:

  ```
  skopeo inspect --config \
  docker://docker.io/library/nginx:1.27.4
  ```

- Copy the nginx:1.27.4 container image from the library repository on the docker.io container image registry to the local OCI image store:

  ```
  sudo skopeo copy \
  docker://docker.io/library/nginx:1.27.4 \
  oci://nginx:1.27.4
  ```

  Mind we are using sudo because the local OCI images store is a system-wide object stored beneath the /var/lib directory tree. Its configuration is defined in the /etc/containers/storage.conf file.

- Copy the nginx:1.27.3 container image from the library repository on the docker.io container image registry to the OCI archive file nginx-1.27.3.tar:

  ```
  skopeo copy docker://docker.io/library/nginx:1.27.3 \
  oci-archive:nginx-1.27.3.tar
  ```

The generated tarball can then be copied to another host. A typical use case is dealing with Docker hosts in an air-gapped environment – in such a scenario, once the tarball is copied on the remote host, we can log in to it and copy the contents of the OCI archive file `nginx-1.27.3.tar` to the local OCI image store:

```
sudo skopeo copy oci-archive:nginx-1.27.3.tar \
oci://nginx:1.27.3
```

This two-step process enabled us to make the image available locally on the Docker host in the air-gapped environment without needing to pull from a registry.

- Display detailed information of the `nginx:1.27.4` and `nginx:1.27.3` container images on local OCI image store:

```
skopeo inspect oci://nginx:1.27.4
skopeo inspect oci://nginx:1.27.3
```

- Copy the to the `nginx:1.27.2` container image from the `library` repository on the `docker.io` container image registry directly into the filesystem, expanding its contents into the `nginx-1.27.2` directory tree:

```
skopeo copy docker://docker.io/library/nginx:1.27.2 \
dir:nginx-1.27.2
```

You can now see the container's image extracted structure by running `find nginx-1.27.2`. As you can see we extracted every single container image's layer: you can have a close look at any of them using tar. For example: `tar tfv nginx-1.27.2/81815610209d2f9b0 4f5a15fd617d8d30974efdbf8e676c1cefee754becc5819`.

- Publish the locally available `nginx:1.27.4` container image to `3rdparty` repository on the `reg.t1.carcano.corp` registry.

```
skopeo copy oci://nginx:1.27.4 \
docker://reg.t1.carcano.corp/3rdparty/nginx:1.27.4
```

- Synchronize every single tag of the `fooapp` container image from the `carcano` repository in the `reg.t1.carcano.corp` container image registry to the `carcano` repository in `reg.carcano.ch` customer-facing container image registry (this is just a fictional example, since we don't have `reg.carcano.ch` in our lab).

  ```
  skopeo sync --src docker --dest docker \
  reg.t1.carcano.corp/carcano/fooapp \
  reg.carcano.ch/carcano
  ```

- Promote from the `t1` container image registry (testing environment) to the `p1` (production environment) the container image `nginx:1.27.4` belonging to the `3rdparty` repository (this is just a fictional example, since we don't have `reg.p1.carcano.ch` in our lab).

  ```
  skopeo copy \
  docker://reg.t1.carcano.corp/3rdparty/nginx:1.27.4 \
  docker://reg.p1.carcano.corp/3rdparty/nginx:1.27.4
  ```

- Delete from the `3rdparty` repository on the `reg.t1.carcano.corp` container image registry the container image `nginx:1.27.4`

  ```
  skopeo delete \
  docker://reg.t1.carcano.corp/3rdparty/nginx:1.27.4
  ```

- After pulling the `nginx:1.27.2` container image from the `library` repository on the `docker.io` container image registry, copy it from the local container image cache to the local OCI image store – mind the use of `sudo` in both statements.

  Be wary that this operation makes sense only when using `podman`, since with `docker` there is no personal container storage:

  ```
  sudo podman pull library/nginx:1.27.2
  ```

```
sudo skopeo copy \
containers-storage:docker.io/library/nginx:1.27.2 \
oci://nginx:1.27.2
```

Running Containers

While familiarizing yourself with containers, it is very likely to start by running already existing container images rather than start building your own, so we start our guided tour by learning how to operate them.

Operating Containers

For your convenience, Table 28-2 summarizes the most used statements for operating containers using existing container images.

Table 28-2. *Docker container actions*

Action	Description

`create`	Create a container – for example:
	`docker create --name mariadb \`
	`-p 3306:3306 \`
	`-e MYSQL_ROOT_PASSWORD=grimoire \`
	`-v mariadb_data:/var/lib/mysql \`
	`mariadb:latest`
	Creates a MariaDB container called `mariadb`, exposing the container's port `3306` as `3306` on the host. The MariaDB's `root` user's password is set by using the `MYSQL_ROOT_PASSWORD` variable.

(*continued*)

Table 28-2. (*continued*)

Action	Description

`run`	Create and immediately start a container. As an example, let's assume you have the `bar_db` database on `mysql.carcano.corp`, and you want to connect to it as `foo_user`. This can be easily done by running a container as follows:
	`docker run -it --rm mysql:latest \`
	`mysql -h mysql.carcano.corp -u user \`
	`-p bar_db`
	This statement spins up a containerized mysql client and connects it to the `bar_db` database on the `mysql.carcano.corp` host.
	In this specific use case we use the `--rm` option to ask `docker` to remove the container once terminated (the default is to keep them so to be able to start them as necessary), whereas `-it` means attach to the container interactively and set a teletype. The other ones are normal options of the `mysql` command-line utility (`-h` the database host,`-u` the connecting user, `-p` ask for the password interactively)

`start`	Start a container – for example: `docker start mariadb`
`ps`	List the running containers. If used with the `-a` option, it lists any available containers, including the not running ones. For example: `docker ps -a`
`top`	Inspect the processes inside container – for example: `docker top mariadb`
`attach`	Attach the terminal to the stdout / stderr / stdin of a running container – for example: `docker attach mariadb` Once attached, if the running container is an interactive shell, then you can run statements through it. If instead the running container is just running a process, such as an NGINx instance, you can just send signals to the process, for example, pressing `CTRL+C`. To detach, hold `CTRL`, then press `p` and `q`. Mind not to attach to containers that were not created with a tty (`-t` option) or you will not be able to detach.

(*continued*)

Table 28-2. (*continued*)

Action	Description
`logs`	Display the stdout of a container – for example: `docker logs mariadb` It works with containers following the best practice of logging to standard output. This approach allows for easier log management and integration with logging systems, as the logs can be easily accessed and monitored.
`exec`	Execute a command inside a container – for example: `docker exec -ti mariadb bash`
`kill`	Send a POSIX signal to a container – if the `--signal` parameter is omitted, then it sends SIGKILL, immediately terminating the container: `docker kill --signal=SIGTERM mariadb`
`restart`	Restart a running container – for example: `docker restart mariadb`
`stop`	Stop a container – for example: `docker stop mariadb`
`export`	Export a stopped container into a tarball archive. For example: `docker export mariadb -o mariadb.tar` Mind that this process strips container's metadata and flattens every container image's layer into a single one, storing it into the exported tarball.
`import`	Import a tarball generated from `docker export` or from a filesystem snapshot into a new container image. For example: `docker import mariadb.tar mariadb`

(*continued*)

Table 28-2. (*continued*)

Action	Description

`inspect`	show detailed information of a container – for example:
	`docker inspect mariadb`
	By providing the `-f` option, it is possible to extract specific information by using the Go templating syntax. For example:
	`docker inspect -f '{{.State.Status}}' mariadb`

`rm`	Delete an existing stopped container: `docker rm mariadb` if necessary you can force the deletion by passing the `-f` option.

`cp`	Copy filesystem objects from the local filesystem to container and vice-versa. For example:
	`docker cp docker-registry:/entrypoint.sh`
	`docker cp nginx.conf docker-registry:/home`

Operating Container Images

It is of course necessary to learn also how to manage existing container images. Table 28-3 summarizes the syntax for the most used tasks on this topic.

Table 28-3. *Docker container image actions*

Action	Description
`login`	Log in to a container image registry and cache the credentials for further requests. For example: `docker login reg.t1.carcano.corp`
`logout`	Remove the cached credentials. For example: `docker logout reg.t1.carcano.corp`

(*continued*)

Table 28-3. (*continued*)

Action	Description
`search`	Locate an image on a registry. For example: `docker search nginx` Mind that since this statement relies on the deprecated `v1` API, its functionality may be limited or not as effective when working with some container image registries.
`pull`	Pull an image from a container image registry. For example: `docker pull nginx:1.27.4`
`images`	List the available local container images. For example: `docker images`
`rmi`	Delete a local container image. For example: `docker rmi mysql:latest`

`tag`	Tag local container images. When tagging, most of the time the tag includes also the container image registry endpoint URI, so to be able to push the image to it as necessary.
	For example:
	`docker tag nginx:1.27.4 \`
	`reg.t1.carcano.corp/3rdparty/nginx:1.27.4`

`push`	Upload an image to a container image registry. For example:
	`docker push \`
	`reg.t1.carcano.corp/3rdparty/nginx:1.27.4`
	Mind that this requires having already cached credentials for the target registry, otherwise you must first run `docker login`.

`history`	Display a local container image's ancestors. For example: `docker history nginx:1.27.4`
`inspect`	Display low level information about a local container image. For example: `docker inspect nginx:1.27.4`

(*continued*)

Table 28-3. (*continued*)

Action	Description
`save`	Save a local container image to a tarball archive file. For example: `docker save -o nginx-1.27.4.tar nginx:1.27.4` It is worth to mention that in this case the tarball archive file retains every single container image layer, without flattening them, and it retains metadata too.
`load`	Import a local container image from a tarball archive previously generated by `docker save`. For example: `docker load -i nginx-1.27.4.tar` The imported container image has all its layers and its own metadata.
`prune`	Cleanup unused local container images. For example: `docker image prune -a`

Checkpoint and Restore

Both Docker and Podman provide good integration with **CRIU** (Checkpoint/Restore In Userspace), enabling users to checkpoint the container state either saving in the system or dumping it to a file.

Example use cases addressed by this feature are as follows:

- Perform hardware maintenance requiring a machine shutdown, resuming the running containers to the state they were after the maintenance window.
- Move running containers from a containerization host to another, resuming the running containers to the state they were after the maintenance window.
- Dump the container along with its state to an archive that can be used for forensics.
- Roll back a container to a previous point in time.

The workload that benefits most from this feature is time-consuming batches: services are indeed typically stateless and deployed with multiple replicas on different containerization hosts, so it is very unlikely needing to be checkpointed and migrated.

Note If you are checkpointing for moving the container to another host or to resume after a system restart, remember to run the statement avoiding options that leave the container running. Conversely the restored state will be before the last container's state, potentially causing unpredictable side effects.

Docker

Note Checkpointing in Docker is still an experimental feature: I'm showing it because it is very likely they will release a stable interface sooner or later, but in the meantime consider this just as a learning exercise that may even not work with your Docker version, so please don't try it on a production system.

Since creating checkpoints and restoring them is still an experimental feature in Docker, it is not enabled by default: to use it you must enable experimental features by adding `"experimental": true` to the `/etc/docker/daemon.json` file.

For example:

```
{
  "experimental": true
}
```

Then restart the `docker` service to apply the changes:

```
sudo systemctl restart docker
```

In addition to that, you must install the `criu` RPM package:

```
sudo dnf install -y criu
```

Now that everything is properly set up, let's simulate a batch workload by spinning up a container running an infinite loop print the iteration number at each iteration:

```
docker run -d --name counter alpine \
sh -c 'i=0;while true;do echo "$i"; i=$((i + 1));sleep 5;done'
```

Let's then attach to the container's stdout:

```
docker logs -f counter
```

and wait 20–30 seconds to have a few iteration numbers logged, then press CTRL+C to detach from the log.

Then, create the checkpoint called pt1 as follows – mind this terminates the running container:

```
docker checkpoint create counter pt1
```

Note If necessary, you can leave the container running after the checkpoint creation by using the `--leave-running` option. Use cases are creating a checkpoint for rolling back a process to a previous point in time. Another use case is forensics: in this case you must also use the `--checkpoint-dir` option to specify where to dump the checkpoint.

Now, let's simulate a hardware maintenance operation by restarting the machine. After logging in again, we start the container counter again, but conversely from a normal start, we pass the name of the checkpoint using the `--checkpoint` option, causing Docker to resume the state in that checkpoint.

```
docker start --checkpoint pt1 counter
```

Note During testing on an ARM-based Docker host, the command failed to complete and raised a network namespace error. As noted earlier, this is still an experimental feature.

If you now have a look at the logs, it resumes from where it was, increasing the counter from the last processed number:

```
docker logs -f counter
```

Podman

Note Podman supports checkpointing with CRIU, but since this requires privileged capabilities, it works only with containers started as the `root` user.

As an example, use case, this time we simulate migrating a workload from a host running Podman to another one.

As the `root` user, let's start by creating a container the same way we did in the Docker example:

```
podman run -d --name counter alpine \
sh -c 'i=0;while true;do echo "$i"; i=$((i + 1));sleep 5;done'
```

Let's then attach to the container's stdout:

```
podman logs -f counter
```

and wait 20–30 seconds to have a few iteration numbers logged, then press `CTRL+C` to detach from the log.

Now, let's dump the checkpoint into a file as follows:

```
podman container checkpoint counter -e /tmp/checkpoint.tar.gz
```

Mind this operation stops the running container.

Note For the sake of completeness, be wary that the kernel does not support pre-copy checkpointing on ARM-based systems.

Then, by using `scp`, copy `/tmp/checkpoint.tar.gz` to the target host.

```
scp /tmp/checkpoint.tar.gz otherhost.carcano.corp:/tmp
```

Once done, log in to the target host and restore the container as follows:

```
podman container restore --import=/tmp/checkpoint.tar.gz
```

If you now have a look at the logs, it resumes increasing the counter from the point it was

```
podman logs -f counter
```

Note Same as with Docker, also with Podman you can save the running state of containers and restore them after rebooting the system: just run without any options `podman container checkpoint`, shutdown the system and operate your maintenance, start the system again, and run `podman container restore`.

Creating Container Images

Despite there being a lot of very good off-shelf container images often provided by the official software vendors, sometimes it is necessary to know how to create custom container images specifically fitting our needs.

Use cases are not limited to containerizing custom applications, but also adding features to existing off-shelf container images to address compliance requirements or to better fit corporate standards.

A good and easy starting point for learning how to deal with creating custom container images is studying the old CentOS 7 Apache `httpd` container image building manifest.

As an exercise, we will update it to work with Rockylinux 9, showing the process first with Docker and then with Podman.

Since the resources to be used in this example are in the `httpd` subdirectory of this chapter's workspace, change to it:

```
cd ~/devsecops-pillars/ch28/httpd
```

Then, create a file called `Dockerfile` with the contents of https://hub.docker.com/r/centos/httpd/dockerfile:

Let's have a closer look at it:

```
FROM centos:7
```

The `FROM` directive requests to import every container image layer from the `centos:7` image.

Since we are modernizing it, replace `centos:7` with `rockylinux:9`.

The next directives are:

```
MAINTAINER The CentOS Project <cloud-ops@centos.org>
LABEL Vendor="CentOS" \
      License=GPLv2 \
      Version=2.4.6-40
```

Update them with your actual data. By the way, mind that the `MAINTAINER` directive is deprecated and must be replaced with a `LABEL`.

The overall change must look like as follows:

```
LABEL Maintainer="Carcano SA <ci@carcano.corp>" \
      Vendor="Carcano SA" \
      License=GPLv2 \
      Version=2.4.62-4
```

Just mind adjusting the `Version` label to match the actual apache `httpd` version installed on the container.

The above statements are used to add metadata about the maintainer (`MAINTAINER` directive) as well as a few labels (`LABEL` directive).

Then there is the `RUN` statement:

```
RUN yum -y --setopt=tsflags=nodocs update && \
    yum -y --setopt=tsflags=nodocs install httpd && \
    yum clean all
```

As you can see, the `RUN` directive contains several `yum` statements to update the Linux system within the container and install the `httpd` RPM package.

When writing your own Dockerfiles, mind the trick of using the logical AND (`&&`) continuing the `RUN` directive on the next line using the backslash character (`\`).

This trick causes every statement to be part of the same pipeline within the same `RUN` command: if any of the yum statements fail, then the whole `RUN` command will be marked as failed causing the whole build to fail.

Another trick you must be wary about, if the pipeline in the `RUN` directive involves piping the output of any command, is setting the `pipefail` shell option.

When it is not set, the pipeline exit status is the exit status of the last command: this hides failed commands from the overall outcome, potentially causing unpredictable effects in the built container image. To set it, just begin the `RUN` pipeline by `set -o pipefail;` .

Note Since each RUN directive causes a new layer to be added to the container image, the best practice is to consolidate as many statements as possible within the same RUN directive to avoid the creation of unnecessary container image layers.

Continuing with the Dockerfile, we get

```
ADD run-httpd.sh /run-httpd.sh
```

The ADD directive is used to add the specified contents to the container image. In this case, it copies the `run-httpd.sh` file from the current directory of the build host into the container's root directory (`/`):

Note Mind also that the ADD directive creates an additional layer.

Then we have one last RUN statement:

```
RUN chmod -v +x /run-httpd.sh
```

The above RUN directive is used to assign execute permission to the copied file (`run-httpd.sh`).

Note Mind the number of added layers: so far, we counted two RUN and one ADD directives, so this Dockerfile adds three new layers to the container image it is derived from.

So far, the `Dockerfile` contains only directives operating a build time for creating the container.

The next two directives are instead processed at runtime:

```
EXPOSE 80
```

EXPOSE exposes the container's port 80 on the host running the container: this enables the outside world to access the HTTPd service running inside the container.

```
CMD ["/run-httpd.sh"]
```

The CMD specifies the command to run at container startup – run-httpd.sh in this case.

Docker

When dealing with Docker, container images are created by running the docker build statement.

When running this statement, Docker just looks into the specified directory for a file called Dockerfile and processes it.

To complete this lab, let's launch the build process by running

```
docker build \
-t reg.t1.carcano.corp/carcano/httpd-rocky9:2.4.
```

Mind how the -t parameter is used to provide the container image tag to apply after the build.

You may now run the container you built, perform some tests, and, if you think it is ready to be published, push it to the container image registry.

Podman (buildah)

Conversely from Docker, the podman command-line tool does not have the build action, delegating the build process to a separate tool called buildah.

Note The name "Buildah" is rumored to come from Dan Walsh's Boston accent, where "builder" may have sounded like "buildah". If this is true, it highlights the importance of clear written communication, as I mentioned in the first chapter of this book. For context, Dan Walsh is known for leading significant projects like SELinux and Podman – kudos to him!

Since the buildah command-line tool is provided by a dedicated RPM package, we must install it as follows:

```
sudo dnf install -y buildah
```

Buildah processes a file called Containerfile, which uses the same syntax as a Dockerfile.

As an exercise to play with `buildah`, let's create the `Containerfile` hard link to the `Dockerfile` file:

```
ln Dockerfile Containerfile
```

We can now launch the build statement as follows:

```
buildah build -t reg.t1.carcano.corp/carcano/httpd-rocky9:2.4
```

Note By default `buildah` generates Docker images but, if necessary, you can ask the creation of an OCI image by adding the `--format oci` option.

Networking and Volumes

Both Docker and Podman provide networking as well as storage volume facilities:

- The `docker network` (or `podman network`) commands allow you to create and manage container networks.

  The default network type is a bridge network, which enables containers to communicate each other and with the outside world through Network Address Translation (NAT).

  You can also create custom bridge networks to isolate groups of containers that need to work together while keeping them separate from other containers on different networks.

- The `docker volume` (or `podman volume`) commands enable the creation and management of storage volumes.

  These volumes can be mounted into containers to provide persistent storage, which is essential for data that needs to persist beyond the lifecycle of a container.

  Docker volumes can be stored on the host filesystem or backed by network file systems (NFS) or other storage solutions.

Note Be aware of Podman's daemonless design. When you run containers as the `root` user, Podman behaves similarly to Docker, allowing containers to communicate with each other easily. However, when you run containers as a non-root user (rootless containers), they are isolated from each other by default, even if they are on the same network. To enable communication between them, you need to set up special configurations, such as binding their ports on the host.

The syntax of both commands is very well documented in Podman and Docker documentation, so we won't repeat them here.

I highly recommend having a look at their original documentations:

Docker

- Network: https://docs.docker.com/engine/network
- Volumes: https://docs.docker.com/engine/storage/volumes

Podman:

- Network: https://docs.podman.io/en/stable/markdown/podman-network.1.html
- Volumes: https://docs.podman.io/en/stable/markdown/podman-volume.1.html

Mind that it's quite unlikely you have to manually run these commands: deployment spun up using `docker compose` or `podman compose` statements automatically take care of setting up networking and volumes as necessary indeed.

For example, the docker registry deployment we run created the `registry_default` network.

You can verify this by running

```
docker network ls
```

Note Don't neglect studying these commands: they are very important when dealing with troubleshooting, which is one of the tasks a DevOps and DevSecOps professional is supposed to manage.

Cleanup

Before moving to the next chapter, perform the cleanup of the containers we created and their related container images, since we won't need them anymore:

```
cd ~/devsecops-pillars/ch28/registry
docker compose down
docker rmi alpine:latest nginx:alpine registry:2
docker rmi reg.t1.carcano.corp/carcano/httpd-rocky9:2.4
```

Summary

This chapter explored the use of Docker and Podman for operating containers and developing custom container images.

After setting up a suitable workspace with a local container image registry, we introduced the fundamentals of spinning up containers and interacting with container image registries, experimenting also with tools like Skopeo and CRIU. To complete the skill set, going beyond the simple use case of using third-party container images, we also experimented with the development of a custom one, discussing and modernizing the old Apache HTTPd container image for CentOS.

Proficiency in Docker and Podman is crucial for DevOps and DevSecOps professionals, as these tools streamline the creation, management, and deployment of containerized applications. This capability allows for efficient resource utilization and scalability, both of which are essential in modern software development.

Mastery of these technologies enables professionals not only to optimize workflows, improve collaboration, and ensure consistency across development and production environments; it also enables enhanced security and compliance, thereby facilitating the implementation of best practices in DevOps and DevSecOps.

As organizations increasingly embrace containerization, expertise in these tools becomes a vital asset for professionals seeking to enhance CI/CD pipelines and maintain robust security postures.

The concepts covered in this chapter should adequately equip you as a professional to tackle the daily challenges that may arise when working with Docker and Podman.

CHAPTER 29

Deploying Pulp3 Using Docker Compose

Both Docker and Podman provide facilities for deploying and managing containers by specifying the `compose` action – we already hit upon using the `compose` action when, in the previous chapter, we deployed a solution running both NGINx along with the Docker registry containers. That enabled us to quickly spin up a container-based solution, still without having skills on container technologies, by simply running a Docker compose statement.

Being skilled in both the use of the command line as well as the `compose` action is essential for DevOps and DevSecOps professionals: as using the compose action simplifies the creation and management of applications in isolated environments, it is very likely they will ask you to write compose manifests or to support developers and QA on this.

As an example use case of using `docker compose`, this chapter shows you how to deploy Pulp3 using the official Docker compose manifest provided by the Pulp3 project.

Demonstrating how to deploy Pulp3 and a quick overview of its usage (we will use it also in other chapters) is certainly a valuable plus: Pulp3 is indeed a very interesting software which can be very easily used as a brick in CI/CD to manage software repositories, enabling teams to efficiently distribute and update packages while maintaining version control and compliance.

Docker and Podman Compose

When using either Docker or Podman with the `compose` action, both tools load deployment information from a YAML-formatted manifest file that has a similar syntax.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_29

This manifest file, called `docker-compose.yml`, not only defines the containers to be deployed but also specifies their requirements, such as networks and volumes, allowing for fine-grained customization using environment variables as needed.

Note When using Podman, the networking behavior may vary if the containers are started by users other than `root`, as Podman is designed to support rootless container execution.

Example Deployment: Pulp3

As anticipated, to experiment with a more complex example than the one of the previous chapter, we will deploy Pulp3 using the official manifest file available on Github.

What Is Pulp

Pulp is an open-source repository manager, which is an application for managing various kinds of repositories including software packages, making it easier to distribute and manage software across various environments.

This enables teams to efficiently handle repositories of various types. Pulp is not limited to container images, RPMs, Deb, and Python packages: it can even be used as a local repo for Ansible's Roles and Collections, making it versatile for different scenarios.

All these features make it an essential component for maintaining consistency and reliability in software deployments.

Designed to support large repositories and scale as the needs of the organization grow, it is a valuable tool due to its ability to streamline package management and the facility of integrating with CI/CD pipelines, enhancing automation and reducing manual intervention.

Its latest evolution to Pulp3 introduced enhancements like improved performance, a more flexible architecture, and better support for modern software distribution needs.

Since the official Pulp3 repository on Github provides the compose manifest file to run it, this gives us the opportunity not only to see container deployments in action, but also to easily have a go with this very nice piece of software.

Deploy Pulp3

First things first, we obviously must clone the official Pulp3 container image project from GitHub.

Change to the user's home directory:

```
cd ~
```

then run

```
git clone https://github.com/pulp/pulp-oci-images.git
```

The repository contains all the necessary resources to build the container images required to deploy Pulp3.

Since we are interested only in the Docker compose manifest file, let's change to the `images/compose` directory:

```
cd ~/pulp-oci-images/images/compose
```

Now, with an editor such as `vi`, open the `compose.yml` file. Besides having a look at it, we will also need to tweak a few settings.

Note Spend some time having a close look at the `compose.yml` file, learning the syntax and understanding the logic. You might find it useful to check the official Docker Compose documentation at https://docs.docker.com/reference/compose-file.

Since we are in a lab, we don't need all the redundancies that are necessary in a production environment. To save resources, we reduce the number of replicas by changing each `replicas` attribute from two to one.

In a real-life scenario, we would also change the following passwords:

- `POSTGRES_PASSWORD`: This is the password of the PostgreSQL user who owns the Pulp3 database.
- `PULP_DEFAULT_ADMIN_PASSWORD`: This is the Pulp3 administrative user account. Note that if you change this, you must also update the `assets/settings.py` values accordingly.

Since we are in a lab, leave both as default.

Let's spin up the deployment by running

```
docker compose up -d
```

If you are using Podman, the statement to run is

```
sudo podman compose up -d
```

Similarly, when necessary, you can take down the deployment by running `docker compose down -v` or `sudo podman compose down -v`.

Please be wary that taking down means getting rid of the deployment.

If you need to stop the deployed containers, use `docker compose stop` or `sudo podman compose stop`; to start them, use `docker compose start` or `sudo podman compose start`.

Note Don't take this deployment down we are going to use this deployment in other labs in other chapters. In addition, please **remember to always stop the Pulp deployment before shutting down the VM** by running `docker compose stop` or `sudo podman compose stop` to avoid the risk of corrupting the PostgreSQL database.

The complete spin-up can take some minutes, so don't worry if everything doesn't come up shortly.

If using Docker, when the process completes, the output should look as follows, but it can of course have slight differences depending on your Docker version:

```
✓ Network compose_default                              Created
✓ Volume "pg_datadev"                                  Created
✓ Volume "pulpdev"                                     Created
✓ Volume "redis_datadev"                               Created
✓ Container compose-postgres-1                         Healthy
✓ Container compose-redis-1                            Healthy
✓ Container compose-set_init_password_service-1        Exited
✓ Container compose-migration_service-1                Exited
✓ Container compose-pulp_worker-1                      Started
✓ Container compose-pulp_content-1                     Healthy
```

```
✓ Container compose-signing_key_service-1          Exited
✓ Container compose-pulp_api-1                     Healthy
✓ Container compose-pulp_web-1                     Started
```

Podman is less verbose at deployment time: if you want to check the running containers, you must run `podman ps`.

As you see, this deployment is an entire ecosystem made of several containers.

Table 29-1 describes each of them.

Table 29-1. *Containers in the Pulp3 deployment*

Container	Description
`postgres`	Runs the PostgreSQL database used by Pulp3.
`redis`	Runs Redis, an in-memory data store used as a message broker and caching layer.
`set_init_password_service`	Sets the initial admin password for the Pulp application.
`migration_service`	Handles database migrations, running the necessary migration scripts to prepare the database for use. The purpose is ensuring that the database schema is up to date with the latest changes in the Pulp application.
`signing_key_service`	Sets up the signing key service, which is essential for signing packages managed by Pulp.
`pulp_content`	Manages the content within Pulp3, handling the storage, retrieval, and organization of software packages, ensuring that they are available for distribution.
`pulp_api`	Runs the RESTful API for interacting with the Pulp3 application. It allows users and other services to perform operations such as uploading, downloading, and managing software packages.
`pulp_worker`	Runs the Pulp worker processes, which handle the asynchronous background tasks such as content synchronization and processing.
`pulp_web`	Runs the web interface for Pulp3, providing a user-friendly way to allow users to manage repositories, view content, and perform administrative tasks through a graphical interface.

Log In to the Pulp3 Web UI

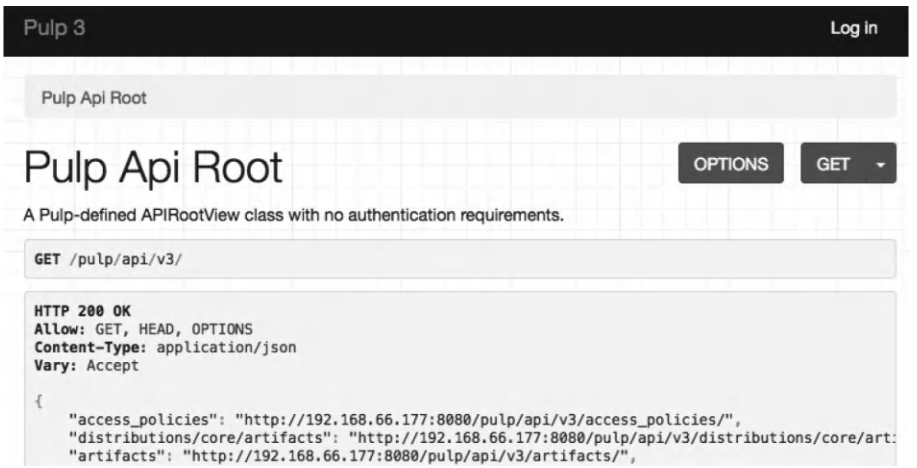
Once the deployment is up and running, we can have a look at the Pulp3 web UI by opening the URL in a web browser. If you open the browser on the same machine running the Pulp3 deployment, the URL to open is

`http://localhost:8080/pulp/api/v3/`

Conversely, if you are connecting from a different machine than the one running the Pulp3 container, replace localhost with that machine's IP address.

Note To find the IP address of your lab's virtual machine (VM), run the command `ip -4 a` in the VM's terminal. This will list the IPv4 addresses associated with its network interfaces. If you launched the VM using the `Vagrantfile` provided with the book, the relevant IP address is typically assigned to the `eth1` interface. If you're unsure which IP to use, check for the one that shares the same subnet as your host machine's IP address. You can verify this by comparing network prefixes (in my own scenario, both starting with `192.168.66.x`).

The Pulp3's API UI looks as follows:



Note Do not delete this deployment: we will exploit it in other labs in the next chapters.

Install the Pulp CLI

Since Pulp3 is one of the bricks of the labs in this book, we must also install the Pulp3 command-line utility: this is a Python module providing a command-line tool that can be used to easily set up and manage repositories in the Pulp3 instance we just deployed.

Since in Chapter 22 we created the `build` user with the specific purpose of running build-related and continuous integration tasks, we must first switch to that user:

```
sudo su - build
```

We can now install the `pulp-cli` Python module:

```
python3 -m pip install pulp-cli[pygments]
```

To avoid manually having to specify the Pulp3 instance's URL and authentication data each time we run a Pulp3 CLI statement, we create a configuration file storing these data.

Just run

```
pulp config create --username admin --base-url \
 http://localhost:8080 --password password
```

Since we are running the statement on the same host running the Pulp3 containers, we specify `localhost` as the base URL. As for the `admin` user's password, it must match the `admin` user's password specified in the `compose.yml` file, which is `password` by default.

The outcome of this command is the creation of the `~/.config/pulp/cli.toml` configuration file.

Note Since the configuration file contains also the credentials used to authenticate to the Pulp3 instance, you may want to tighten security permissions by running `chmod 0600 ~/.config/pulp/cli.toml`.

Create a Repository in Pulp3

We are now ready to create our first repositories for storing RPM packages.

In this example we are about to create these two RPM repositories:

- `carcano-test-el9-x86_64`
- `carcano-test-el9-aarch64`

We will use them later to publish our custom RPM packages, such as the `fancy-monitor` application.

As the `build` user, let's create them by running

```
pulp rpm repository create --name carcano-test-el9-x86_64
pulp rpm repository create --name carcano-test-el9-aarch64
```

If everything is properly working, each of the above statements receives a JSON document as a reply describing the just-created RPM repositories.

We can of course list the available repositories by command line by running

```
pulp rpm repository list | jq ' map( .name)'
```

The output is

```
[
  "carcano-test-el9-aarch64",
  "carcano-test-el9-x86_64"
]
```

Note For your convenience, the Pulp CLI documentation is available at https://pulpproject.org/pulp-cli/.

Summary

As containerization grows in modern software development, Docker and Podman empower professionals to build, manage, and scale applications more effectively, improving operational efficiency and maintaining robust security postures.

This short chapter demonstrated how easy it is to deploy even quite complex container-based solutions using Docker Compose or Podman Compose.

Proficiency in using these tools' `compose` feature is essential for DevOps and DevSecOps professionals, as it facilitates the management of multi-container applications. By allowing users to define and run complex applications with multiple services in a single YAML file, Docker Compose and Podman Compose streamline deployment, manage dependencies and configurations, and ensure consistency across environments – from development to production. They can be effectively used also as a convenient support to workflow automation, optimizing CI/CD pipelines and accelerating software delivery while enhancing collaboration between development and operations teams.

As an example of a complex solution, we deployed Pulp3: understanding how to install and operate Pulp3 is crucial for DevOps and DevSecOps professionals, as it automates content management to ensure the right software versions are available when needed. Moreover, Pulp3 also supports security best practices through content lifecycle management, including version control and access policies, which integrate security into development and deployment processes.

In organizations adopting microservices and containerization, proficiency in Pulp3 helps to optimize CI/CD pipelines and to maintain robust security postures. Overall, knowledge of Pulp3 equips professionals to manage software distribution, streamline workflows, and enhance team collaboration.

PART X

Service Protocols

In a distributed architecture, a DevOps and DevSecOps professional must know how to govern and secure the core interfaces that connect microservices. Advanced expertise requires the technical ability to audit contract compliance using OpenAPI, Swagger, and WSDL specifications for both REST and SOAP web services. Mastering these interfaces enables you to generate robust Python libraries directly from API contracts and deploy service mocking techniques to simulate complex backend behaviors—ensuring your pipelines validate integrations efficiently, safely, and with minimal computing overhead. Crucially, this programmatic approach eliminates the massive infrastructure costs and resource consumption of spinning up expensive, complex live backend environments for basic interface validation.

CHAPTER 30

SOAP and REST

In Chapters 24–26, we explored essential protocols that every DevOps and DevSecOps professional must be familiar with, establishing a foundation for understanding the communication standards that underpin modern applications.

This chapter advances the discussion by focusing on two widely used application protocols for implementing web services: REST and SOAP.

Mastery of these protocols is crucial for DevSecOps professionals, as they facilitate the development and integration of APIs, which are vital for enabling seamless communication between services.

Additionally, we will learn how to mock services using Gatehill's Imposter, allowing professionals to simulate the behavior of real APIs and conduct effective component testing without relying on actual backend services.

Note The labs in this chapter are based on Gatehill's Imposter `4.6.8`.

The ability to mock services not only enhances testing efficiency but also contributes to more reliable software delivery. Collectively, these skills are integral to fostering a robust DevOps and DevSecOps practice that prioritizes security, efficiency, and collaboration.

Note Before going any further, change to this **chapter's workspace directory** by running `cd ~/devsecops-pillars/ch30`.

Simple Object Access Protocol (SOAP)

SOAP (Simple Object Access Protocol) is a protocol for exchanging structured information in the implementation of web services.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_30

It is primarily a stateless protocol, although it is possible to use it in a stateful manner in certain contexts.

SOAP relies on XML documents delivered over HTTP or HTTPS, but it can also operate over other protocols such as SMTP.

A SOAP XML document typically includes the following components:

- **Envelope**: The root element that defines the XML document as a SOAP message
- **Header (optional)**: Contains metadata and information for processing the message
- **Body**: Contains the actual message intended for the recipient
- **Fault (optional)**: Provides information about errors that occurred while processing the message

While a SOAP message does not strictly have to conform to an `XML Schema Definition` (`XSD`), it is considered best practice to adhere to a defined structure.

Note The SOAP 1.1 specification is provided by the following `XSD`: `http://schemas.xmlsoap.org/soap/envelope`.

SOAP is quite a complex protocol, but using it provides several benefits, for example:

- **XML Schema enforcement**: This enables an out-of-the-box and easy check of XML messages' structure, formats, data types, and even values.
- **WS-Security**: A built-in standard for security, enabling secure message transmission, including authentication and encryption.
- **WS-ReliableMessaging**: A built-in standard for reliable delivery, ensuring that messages are delivered even in the case of network failures.
- **WSDL**: Optionally, the SOAP API can be described by a WSDL (Web Services Description Language) document.

Web Services Description Language (WSDL)

WSDL (Web Services Description Language) is used to describe the services offered by a SOAP web service in an XML document called WSDL document.

This document clearly defines the service's operations, the messages involved, and the protocols used for communication.

The valid schema specifications for WSDL are

- WSDL 1.1: `http://schemas.xmlsoap.org/wsdl`
- WSDL 2.0: `http://www.w3.org/2006/01/wsdl`

The typical components in a WSDL document include

- **Types**: Defines the data types used by the web service
- **Messages**: Defines the messages exchanged between the client and the service
- **Port Types**: Defines the operations available in the service
- **Binding**: Specifies the protocol and data format for each operation
- **Service**: Defines the endpoint(s) where the service can be accessed

---

Note You may have a look at the Country Information Service's WSDL. This is the WSDL of a real service providing real-time data about countries, including capital cities, currencies, and geographical information, making it a valuable resource for applications requiring up-to-date country data. You can access it at `http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL`.

Mocking a SOAP Endpoint

As DevOps and DevSecOps professionals, we are required to help the development team to automate the setup of mock servers that simulate the behavior of real APIs for running their component tests without relying on real backend services.

An invaluable tool is certainly Gatehill's Imposter: this is a very easy to set up and deploy mock server supporting the most used protocols, including REST and SOAP.

At its bare minimum setup, while processing client requests and providing responses, it implements the semantic expected by the mocked API returning random values, but it is possible to configure it to provide specific responses to requests, including simulated errors.

In addition to that, Imposter is also available as a container image, so it is very easy to integrate it into a CI/CD pipeline.

The project is very well documented (see https://docs.imposter.sh), and a lot of examples are available in the dedicated Github repository: https://github.com/imposter-project/examples.

Note In this chapter we go through only the basics just to see from where to start, but you are encouraged to go on yourself exploring all their insightful examples. The example repository is currently licensed under Apache-2.0.

Inside the chapter's workspace directory, there are some contents for working with Imposter.

The `soap` directory contains sample contents for mocking up a sample SOAP service. Let's change to it as follows:

```
cd soap
```

Create the `config` directory for the SOAP mock service:

```
mkdir config
```

The easiest way to have a go with Imposter is downloading the very good sample files they provide and use them:

```
BASE=https://raw.githubusercontent.com/imposter-project
EXAMPLE=examples/refs/heads/main/soap/simple
curl -o config/schema.xsd ${BASE}/${EXAMPLE}/schema.xsd
curl -o config/petstore.wsdl ${BASE}/${EXAMPLE}/petstore.wsdl
curl -o config/imposter-config.yaml \
${BASE}/${EXAMPLE}/imposter-config.yaml
```

The files we downloaded are used to simulate their fictional `pet store` SOAP service.

You can easily infer the purpose of each file from their name:

- `schema.xsd` is the SOAP service's XSD schema file.
- `petstore.wsdl` is the WSDL document.
- `imposter-config.yaml` is the Imposter's settings file – in this extremely basic example, it instantiates the soap plugin using `petstore.wsdl`.

Note Imposter considers a settings file any file ending with `-config.yaml` stored in the `/opt/imposter/config` directory.

It's time to really have a go – let's spin up the `imposter` container in the foreground by running

```
docker run --rm -ti -p 8081:8080 \
-v $PWD/config:/opt/imposter/config:Z \
outofcoffee/imposter:4.6.8
```

Log messages are directly printed to standard output.

The following snippet has been trimmed for brevity while preserving the key details of the output:

```
Starting mock engine 4.6.8
Loading configuration file: /opt/imposter/config/imposter-config.yaml
Loaded 7 plugin(s): [js-detector, store-detector, openapi, rest, soap,
js-graal, store-inmem]
No OpenAPI configuration files provided - skipping plugin setup
Using WSDL parser: V2 for: /opt/imposter/config/petstore.wsdl
Discovered 2 schema(s) for WSDL: /opt/imposter/config/petstore.wsdl
Resolved XSD /opt/imposter/config/schema.xsd relative to WSDL path
Adding mock endpoint: SoapBinding -> SOAP /pets/
Mock engine up and running on http://localhost:8080
```

This output shows the engine initializing, loading plugins (such as SOAP and REST detectors), and setting up a mock SOAP endpoint based on the provided WSDL file.

Note While the final line indicates the service is running on port 8080 inside the container, we've mapped it to port 8081 on the Docker host. Therefore, use port 8081 when making calls to the service.

Let's now open another terminal to simulate a SOAP call using `curl`.

First, change to the `soap` subdirectory of this chapter's workspace directory:

```
cd ~/devsecops-pillars/ch30/soap
```

In this example we will make a POST call to the `/pets` endpoint pushing the `document.xml` file.

This file contains the SOAP document used to ask for the details of the item having `3` as `id`.

Open it and have a look at its content, then run the following `curl` statement:

```
curl -X POST -H 'Content-Type: application/soap+xml' \
-d@document.xml "http://localhost:8081/pets/"
```

The above `curl` statement

- Sets the `POST` HTTP action using the `-X` option
- Adds the HTTP `Content-Type: application/soap+xml` header using the `-H` option
- Loads the SOAP document from the file specified by the `-d` option

You will receive the following reply:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
<env:Header/>
<env:Body>
  <urn:getPetByIdResponse xmlns:urn="urn:com:example:petstore">
      <id>3</id>
      <name>string</name>
  </urn:getPetByIdResponse>
</env:Body>
```

If you compare the reply to the `document.xml` file, the SOAP document returned in the response looks pretty much the same, except it contains the `name` tag with the value `string`.

So, instead of returning a realistic value, like the actual name, Imposter returned the field type. This is expected: we created the most basic Imposter configuration possible, without configuring it to return meaningful values.

For your convenience, we also provided the `getpets.py` script: this is an example showing how to run SOAP calls from Python scripts using the `tenacity` package.

Remember that, before trying it, you must install the `python3-tenacity` package by running

```
sudo dnf install -y python3-tenacity
```

Once done, simply run

```
./getpets.py
```

Note When finished with your testing, remember to terminate the Imposter running container by pressing `CTRL+C`.

Representational State Transfer (REST)

REST (Representational State Transfer) API is an architectural style that defines a set of constraints and principles for building and interacting with web services.

It has been designed to enable different software applications, such as web browsers, mobile apps, or even other servers, to interact with resources (data or services), communicating over the Internet using standard HTTP methods.

Conversely from SOAP, it is stateless and supports multiple representations (JSON is the most popular, but it supports XML too).

Relying on HTTP brings several benefits, among them also compatibility with layered architecture, where clients do not need to know whether they are communicating with the end server or an intermediary (like a load balancer or cache).

The HTTP methods most used by REST are

- `GET`: Retrieve a resource
- `POST`: Create a new resource

- `PATCH`: Perform a partial update of an already existing resource
- `PUT`: Update an existing resource
- `DELETE`: Remove a resource

OpenAPI

OpenAPI is a REST protocol improvement enabling a REST API to self-describe itself by providing the OpenAPI document.

This can be either a JSON or YAML document with a standard format enabling both humans and machines to understand the capabilities of a service without needing access to the source code or additional documentation.

The OpenAPI document is derived from the Swagger specifications – the currently supported formats, in order of publications, are indeed Swagger 1.2 and 2.0 and OpenAPI 3.0 and 3.1.

Before running some statements to see these concepts in action, change back to the chapter's workspace directory:

```
cd ~/devsecops-pillars/ch30
```

Then, change to the `rest` directory with the workspace of the REST mock service:

```
cd rest
```

The Gitea online demo instance provides an OpenAPI-compliant document we can use to better understand this document's format.

Let's download it as follows:

```
curl -O https://gitea.com/swagger.v1.json
```

As we said, this is a Swagger document – we can verify this by running

```
jq -r .swagger swagger.v1.json
```

The output reveals it is a Swagger 2.0 document.

An OpenAPI document can automatically generate up-to-date client libraries, server stubs, and automated tests based on its specifications.

Several tools exist for generating client libraries, but OpenAPI Generator is likely the easiest to use. Since container images are available for it, the most convenient method is to run it as a container.

Note In this example we are using OpenAPI Generator `7.14.0`.

Before running it, let's create a directory where to store the generated API client library:

```
mkdir library
```

As an example, let's generate a Python library called `libgitea` from the aforementioned OpenAPI URL.

First as first, let's get the version of the running Gitea instance:

```
curl -s https://gitea.com/api/v1/version | jq -r .version
```

At the time of writing this book, the output is

```
1.25.0+dev-207-ge3887f984e
```

so, the version is `1.25.0`.

We can now spin up the OpenAPI Generator container as follows:

```
docker run --rm -v $(pwd)/library:/local \
openapitools/openapi-generator-cli:v7.14.0 generate \
-i https://gitea.com/swagger.v1.json -g python -o /local \
--additional-properties=packageName=libgitea,\
packageVersion=1.25.0
```

As you can see, the Docker options ensure the container is removed after execution and mount the local `library` directory to `/local` inside the container for output.

The OpenAPI Generator options instead specify the input OpenAPI Document from the provided URL, target `python` as the language, and output the files to the `/local` directory inside the container. Additionally, the properties set the generated package name to `libgitea` with version `1.25.0`.

Just wait a little bit for the container to be pulled and for the `openapi-generator-cli` to start and complete its run.

When finished, change to the `library` directory:

```
cd library
```

Here you will find the `libgitea` directory containing the `libgitea` Python package. You can easily explore the contents by running

```
find libgitea
```

The last step is creating the Python wheel package.
Just run

```
python setup.py bdist_wheel
```

Once the package building process is complete, the Python wheel is available beneath the `dist` directory.

Regarding documentation generation, tools such as Swagger UI and Redoc can create interactive HTML documentation by retrieving the OpenAPI specification file in either JSON or YAML format.

Additionally, some applications offer a built-in Swagger web UI; for instance, Gitea provides this feature. You can explore Gitea's online demo Swagger interface by opening the following link in your browser:

`https://gitea.com/api/swagger`

Like many Swagger-based web UIs, the Gitea Swagger interface includes a "Try it out" feature, which allows users to execute real API calls directly from the documentation interface before they begin writing code.

This functionality enhances the user experience by enabling developers to try and test API endpoints interactively.

Mocking a REST Endpoint

Since Gatehill's Imposter also supports REST API, to see things in action, we will use it the same way as we did for the SOAP API.

The `rest` directory in the chapter's workspace contains sample contents for mocking up a sample REST service.

Let's change to it as follows:

```
cd ~/devsecops-pillars/ch30/rest
```

Now, create the `config` directory for the REST mock service:

```
mkdir config
```

In this example we want to mock Gitea's API.

In the previous lab, when mocking a SOAP endpoint, we made a very basic configuration that simulated just returning answers with the expected structure and formats. By running a simple call, we learnt that, when using defaults, Imposter's responses do not provide realistic values.

This time we are already acquainted with Imposter, so we can create a more complex setup involving also returning custom realistic values.

First, let's move to the `config` directory the Gitea's Swagger document we previously downloaded:

```
mv swagger.v1.json config
```

Upon opening and inspecting the API documentation, you'll notice that it defines numerous endpoints.

For this example, we aim to emulate a call to the `/api/v1/version` endpoint, making the response as realistic as possible.

To do this, create the `config/nodeinfo.json` file by downloading the actual response from a live Gitea instance using the following command:

```
curl -o config/version.json https://gitea.com/api/v1/version
```

Then, create the `config/imposter-config.yaml` configuration file with the following contents:

```
---
plugin: openapi
specFile: swagger.v1.json
resources:
  - path: /api/v1/version
    method: get
    response:
      file: version.json
```

As you can see, it configures Imposter to use the `version.json` file we just created as the response document for replying to the calls to `/api/v1/version`.

Since we completed the setup, we can now spin up Imposter as a container listening on the same port a Gitea instance would listen by default:

```
docker run --rm -ti -p 3000:8080 \
-v $PWD/config:/opt/imposter/config:Z \
outofcoffee/imposter:4.6.8
```

Just as when we previously ran the SOAP mock, Imposter is listening on port `8080` inside the container, but, since port `3000` is Gitea's default port, we mapped it to that port on the Docker host.

To run a test, open another terminal and, using the `curl` command-line utility, call the `/api/v1/nodeinfo` endpoint as follows:

```
curl -s -H "accept: application/json" \
http://localhost:3000/api/v1/version | jq
```

The output is

```
{
  "version": "1.25.0+dev-207-ge3887f984e"
}
```

As you can see, this time you got an answer not only with the expected structure and data formats, but also with realistic values.

For the sake of completeness, we are also providing the `version.py` script in the `~/devsecops-pillars/ch30/rest` directory, which provides an example of running REST calls from Python scripts using the `urllib3.PoolManager` Python module.

You can launch it by just typing:

```
./version.py
```

Note When finished with your testing, remember to terminate the Imposter running container by pressing `CTRL+C`.

Summary

This chapter offers a concise overview of the SOAP and REST protocols, along with key related concepts like WSDL, OpenAPI, and Swagger.

For DevOps and DevSecOps professionals, understanding these elements is essential, as they support effective API management and integration, directly boosting the efficiency and security of software development processes. Proficiency in these protocols and tools promotes seamless service communication, automation, and strong security practices, resulting in better integration, streamlined workflows, and more reliable, maintainable software systems.

To apply these concepts practically, the chapter's labs guided you through generating a Python library using OpenAPI and the `openapi-generator-cli`, as well as mocking REST and SOAP applications with Gatehill's Imposter. These exercises build critical skills for testing and development by allowing you to simulate external services, isolate components for testing without depending on live backends, and detect issues early for quicker feedback and debugging. Using Imposter, you can create realistic test environments to validate applications against diverse scenarios and edge cases, which is especially valuable in CI/CD pipelines for automated quality assurance.

Overall, these mocking techniques foster collaboration between development and operations teams, ensure dependable software delivery, and strengthen security through controlled testing. This capability ultimately supports a more agile and resilient software development lifecycle.

PART XI

Orchestrating Containers

Within distributed architectures, managing isolated container runtimes constitutes only the baseline layer of infrastructure control; DevOps and DevSecOps engineering mandates the capacity to dynamically orchestrate workloads at scale. Enforcing automated horizontal scaling and deterministic resource governance isolates production workloads from baseline failures and runtime degradation. Conversely, deploying unmanaged cluster resources introduces untraceable configuration drift and resource allocation inefficiencies. Modern infrastructure engineering requires a declarative, structured ecosystem in which Kubernetes enforces state definitions, while Helm serves as the standard package management layer to ensure deployment consistency across environments.This architecture becomes paramount with the acceleration of Cloud Repatriation, as enterprises transition critical workloads away from unpredictable public cloud expenditures back to self-hosted, on-premises Linux hardware. While repatriation optimizes operational costs and guarantees absolute data sovereignty, it simultaneously obligates engineers to manage multi-cluster governance without cloud-provider abstraction layers. To bridge this operational gap, centralized open-source management planes like Rancher are deployed to orchestrate on-premises clusters uniformly, enforcing enterprise-grade compliance, role-based access control, and strict security profiles across bare-metal infrastructures. These cohesive solutions transform raw compute resources into a fully governed execution layer capable of sustaining high-availability enterprise workloads.

CHAPTER 31

Kubernetes, Helm, and Rancher

This chapter explores several critical concepts within the Kubernetes ecosystem that are essential for DevOps and DevSecOps professionals.

It provides a comprehensive introduction to Kubernetes and Helm, including instructions on how to install Rancher over K3s as a single container.

Understanding Kubernetes and Helm is fundamental, as they equip professionals with the necessary tools for orchestrating containerized applications and efficiently managing their lifecycle. Rancher simplifies the deployment and management of Kubernetes clusters by offering a user-friendly web-based interface.

For professionals, mastering these technologies is vital, as they not only enhance operational efficiency but also address the security and compliance challenges inherent in modern software development and deployment processes.

This knowledge empowers teams to build, deploy, and manage applications securely and effectively in both cloud-native and on-premises environments.

Ultimately, proficiency in Kubernetes, Helm, and Rancher enables professionals to navigate the complexities of container orchestration while ensuring robust security practices throughout the application lifecycle.

Note Before going any further, change to this **chapter's workspace directory** by running `cd ~/devsecops-pillars/ch31`.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_31

Kubernetes

Conversely from Docker and Podman that are suitable for running manually orchestrated containerized workloads, Kubernetes is a full-featured orchestrator designed to address the day-to-day challenges of running mission-critical services in a resilient fashion.

The project was started by Google in 2014, inspired by their internal container management system called Borg, with the intent of providing a robust platform for automating the deployment, scaling, and management of containerized applications.

Kubernetes reached a stable version (1.0) in July 2015. Since then, it has evolved significantly, gaining widespread adoption within the developer community and enterprises, becoming the de facto standard for container orchestration.

Kubeadm

In the past, installing and upgrading Kubernetes was mostly a manual process, with all the cons deriving from that: it was indeed time-consuming, error-prone, and, because of its structural complexity, definitely a maintenance nightmare.

Being aware of this, they developed and improved the `kubeadm` command-line utility, which is an automation tool that manages most of the task to deploy Kubernetes.

Anyway, despite this being a huge step forward, operating with `kubeadm` can be frustrating, especially for users unfamiliar with Kubernetes architecture. Setting up a production-grade cluster indeed requires careful configuration of networking, storage, and security settings, which can be complex and time-consuming.

Minkube and K3s

To help acquaint with Kubernetes, especially for local development, lightweight solutions such as Minikube and K3s were developed:

- Minikube creates a single-node Kubernetes cluster on a single machine. Designed for local development, it is easy to set up and supports multiple operating systems. Despite clearly not being suitable for production environments, it is excellent for testing and learning.

- K3s is a lightweight version of Kubernetes developed by Rancher, optimized for resource-constrained environments like IoT and edge devices: it simplifies installation and reduces memory footprint by excluding some non-essential features. It is ideal for local testing and small-scale deployments.

Openshift and Rancher

To address enterprise-class needs, Red Hat developed Openshift and Rancher Labs (currently part of Suse) developed Rancher: these two products extend Kubernetes using their own set of Custom Resource Definitions providing a powerful dashboard that is really useful in day-to-day activities.

They hide most of the complexity of the system behind a web UI that anyway provides the very most of the Kubernetes knobs in an intuitive interface that even enables running the Kubernetes command line if necessary.

- Openshift provides built-in CI/CD capabilities, including pipelines and integration with popular tools like Jenkins, GitLab, and Tekton and a streamlined developer experience with features like source-to-image (S2I) builds, which simplify the process of deploying applications directly from source code.
- Rancher is focused on managing multiple Kubernetes clusters across different environments, which can be beneficial for organizations with diverse deployment strategies. It provides a user-friendly interface that makes it easier for teams with varying levels of Kubernetes expertise to manage their deployments although it may require additional setup for CI/CD processes.

A Quick Walkthrough with Kubernetes

It's worthwhile to familiarize yourself with the most fundamental Kubernetes concepts, as they remain essential even when using user-friendly interfaces like Rancher and OpenShift.

The concepts covered in this section are demonstrated through small, practical examples designed for Killercoda (https://killercoda.com), an interactive online platform offering hands-on, browser-based labs in technologies such as Kubernetes and Docker. This approach eliminates the need to install a Kubernetes cluster locally, especially since we'll soon explore installing Rancher.

Killercoda provides SaaS-based Kubernetes environments for your own experiments, with a free tier limiting sessions to one hour—sufficient for these brief labs. You can also explore their pre-configured Kubernetes scenarios at https://killercoda.com/playgrounds/scenario/kubernetes.

Note Since a local Kubernetes instance isn't yet available, all Kubernetes labs in this chapter should be run on Killercoda using the Kubernetes scenario.

Namespaces

Kubernetes groups its workloads into namespaces: these are used to organize resources and can help in applying resource quotas and policies.

When no namespace is specified, the executed statements are applied to the `default` namespace.

Note In addition to Kubernetes namespaces, both OpenShift and Rancher offer another abstraction layer called Projects. These enhance usability and control by allowing you to apply resource quotas at the project level and configure Role-Based Access Control (RBAC).

The Kubectl Command-Line Tool

Kubernetes is operated using the `kubectl` command-line tool: this is a twofold tool, which can be run imperatively to process statements for deploying workload or manage the already running workload or in a declarative way by applying the contents of manifest files.

This utility loads its configuration from the `~/.kube/config` file. Be careful with this file, since it is sensitive: it indeed contains not only the Kubernetes API server's endpoint URL, but also the credentials (such as tokens or certificates) used for authentication to the cluster.

For this reason, `~/.kube/config` must be secured using POSIX permissions or ACLs, granting access only to the users that are eligible to operate Kubernetes.

Note If necessary, it is possible to override the default path of the `kubectl` config file by setting the `KUBECONFIG` environment variable. It is even possible to specify multiple search paths separated by colons.

As an example of imperative deployment using `kubectl`, we deploy the `kubernetes-bootcamp` sample application provided by Google. Just run

```
kubectl create deployment kubernetes-bootcamp \
 --image=gcr.io/google-samples/kubernetes-bootcamp:v1
```

Note If you are not using Killercoda, be wary that the `kubernetes-bootcamp:v1` container image is not available for the `aarm64` architecture.

We can then monitor the deployment process by running

```
kubectl get deployment --watch
```

Wait until the `READY` column is `1/1`, which means 1 out of 1 Pods is fully operational, then press `CTRL+C` to exit the watch mode:

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
kubernetes-bootcamp   1/1     1            1           16s
```

Note Since no namespace was specified, both the preceding `kubectl` statements are applied to the `default` namespace. If you want to apply them to a different namespace, you must specify it by using the `-n` option.

Let's now see the same deployment, but this time performed in a declarative fashion.

In this case, we must create a manifest file with the following contents, saving it as kubernetes-bootcamp.yml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: kubernetes-bootcamp
  name: kubernetes-bootcamp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kubernetes-bootcamp
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: kubernetes-bootcamp
    spec:
      containers:
      - image: gcr.io/google-samples/kubernetes-bootcamp:v1
        name: kubernetes-bootcamp
```

This time, we deploy it to the `carcano` namespace.

First, we create it as follows:

```
kubectl create namespace carcano
```

then, we run the declarative-based deployment by passing the path to the manifest file using the `-f` option:

```
kubectl apply -n carcano -f kubernetes-bootcamp.yml
```

As you can see, the YAML manifest has much more settings compared to the imperative command.

Declarative deployments are indeed the most used ones and are the standard for production and GitOps workflows. They provide greater configuration depth and reproducibility because the entire desired state is defined in manifest files.

Imperative deployments are generally suitable for rapid prototyping, debugging, or one-off tasks. By using the `--dry-run=client -o yaml` flags, they are frequently used to generate base templates, which can then be exported, customized, and re-deployed using a declarative approach

---

Note Describing the `kubectl` commands syntax here would simply be providing a duplicate of the excellent official documentation that would very quickly become obsolete. I strongly suggest you delve into this topic by reading the official `kubectl` documentation at https://kubernetes.io/docs/reference/kubectl/ and the examples at https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands.

Resource Definitions

Kubernetes resources are described by manifests, which contain YAML- or JSON-formatted `Resource Definitions`.

In this book, for space constraints, we quickly discuss only the most commonly used ones, but I strongly advise you to dig into the topic by yourself.

Pod

A Pod is a Resource Definition describing the smallest deployable unit in Kubernetes, consisting of one or more container processes that share the same networking namespace and storage volumes.

In multi-container Pod, Kubernetes may use a `pause` container to maintain the network namespace, allowing the other containers to retain their IP addresses in the event of a crash.

This `pause` container simply runs an endless loop with a sleep command to preserve the network context.

To list running Pods, use the `get` action followed by `pod`.

For instance, to view Pods in the `carcano` namespace, run

```
kubectl -n carcano get pod
```

the output is as follows:

```
NAME                                  READY STATUS RESTARTS   AGE
kubernetes-bootcamp-9bc58d867-6jhqf 1/1  Running   0     2m14s
```

Note When dealing with `kubectl get` statements, no matter if they are related to Pods or other Kubernetes resources, you can add the `-o wide` option to get some more information, or the `-o yaml` option to get a YAML dump of the Resource Definition.

If instead you want to get a human-readable description, use the `describe` action followed by the name of the Pod:

```
kubectl -n carcano describe pod \
kubernetes-bootcamp-9bc58d867-6jhqf
```

Be sure to replace the Pod name with the one from your `get pod` results.

Deployment

A Deployment is a Resource Definition designed to manage the deployment of applications: it contains deployment-specific settings such as the deployment strategy (e.g., recreate or rolling update), the desired number of replicas, and the ability to manage `ReplicaSets`.

You can easily list provisioned deployments by running

```
kubectl -n carcano get deployment
```

The output is as follows:

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
kubernetes-bootcamp   1/1     1            1           12m
```

To see a human-readable description, use the `describe` action followed by the name of the deployment:

```
kubectl -n carcano describe deployment kubernetes-bootcamp
```

As you can see, the last lines of the output of this last command also show you the name of the scaled ReplicaSet generated from the deployment.

Note The `description` action can be generally used with any other Resource Definition.

ReplicaSet

A ReplicaSet is an abstraction that ensures a specified number of `Pod` replicas are running at any given time.

It is created by deriving it from a `Deployment` and represents the deployment's current state at the time it was created or last changed: each time a `Deployment` is changed, a new `ReplicaSet` reflecting the new state is created, and the old one is kept in the history, so as to enable a quick and easy rollback as necessary.

You can easily list the available ReplicaSets as follows:

```
kubectl -n carcano get replicaset
```

The output is

```
NAME                            READY UP-TO-DATE AVAILABLE AGE
kubernetes-bootcamp-9bc58d867  1     1          1         18m
```

To see a human-readable description, use the `describe` action followed by the name of the ReplicaSet:

```
kubectl -n carcano describe replicaset \
kubernetes-bootcamp-9bc58d867
```

As you can see, the last lines of the output of the above command also show you the name of the Pod generated by the ReplicaSet.

Note If you compare the Pod's name and the ReplicaSets's name, you see that they both have a trailing random identifier and that the Pod's name is derived by the ReplicaSet which generated it.

StatefulSet

A `StatefulSet` is a Resource Definition designed specifically for managing applications requiring storage persistence.

This chapter's workspace provides the `statefulset.yml` file, which contains an example manifest of a StatefulSet for deploying PostgreSQL.

Note Please mind that this example completely lacks security concepts, since it's just a didactical example kept at its bare minimum to stay focused only on illustrating StatefulSets, avoiding confusing you with other topics.

Let's deploy it by running

```
kubectl -n carcano apply -f statefulset.yml
```

You can monitor the provisioning process as follows:

```
kubectl -n carcano get statefulset --watch
```

Wait until the `READY` column is `1/1`, which means 1 out of 1 Pods is fully operational, then press `CTRL+C` to exit the watch mode:

```
NAME       READY   AGE
postgres   1/1     9m12s
```

To see a human-readable description, use the `describe` action followed by the name of the StatefulSet:

```
kubectl -n carcano describe statefulset postgres
```

The last lines of the output of the above command also show you the name of the `Pod` generated by the StatefulSet.

As you can see, conversely from Deployments, StatefulSets do not require an intermediate abstraction such as the ReplicaSet.

Instead, Pods are directly derived from the StatefulSet itself, and each Pod is assigned a stable name that includes the instance number.

This behavior enables the generation of predictable Pod names, which allows binding of the Persistent Volume Claims to the correct Pod when scaling or redeploying.

This by the way explains why this Resource Definition is called a StatefulSet.

Persistent Volume Claim

A Persistent Volume Claim (PVC) is a Resource Definition that describes the storage requirements, such as the amount of space, access modes, and optional storage class, required by a Pod.

The PostgreSQL StatefulSet we have just provisioned contain also the Resource Definition of the PVC describing the storage requirements for the PostgreSQL Pod.

We can get it by running

```
kubectl -n carcano get pvc
```

These are the first three columns of the output:

```
NAME             STATUS   VOLUME
postgresql-pvc   Bound    pvc-fa9e4cdf-c76e-4995-b9e7-e2529ea
```

The report reveals that the postgresql-pvc Persistent Volume Claim (PVC) is bound to the pvc-fa9e4cdf-c76e-4995-b9e7-e2529ea Persistent Volume (PV).

Persistent Volume

A Persistent Volume (PV) is a Resource Definition describing and instantiating a storage resource.

The Volume can be backed by various storage types, including local storage, NFS, iSCSI, and FibreChannel.

We can display the available PV by running

```
kubectl get pv
```

Note Here we are omitting the namespace because Persistent Volumes are global objects.

Since the output's length is too large for fitting this book's page, below I'm reporting only a few columns interesting to be discussed, cropping the contents as well:

```
NAME               RECLAIM POLICY STATUS CLAIM
pvc-fa9e4cdf-c76e Delete         Bound  carcano/postgresql-pvc
```

As you can see, the `pvc-fa9e4cdf-c76e-4995-b9e7-e2529ea` PV is `Bound` to the `postgresql-pvc` PVC in the `carcano` namespace.

The `RECLAIM POLICY` column shows that this PV is using the `Delete` reclaim policy. Each PV is indeed subject to a reclaim policy, which is triggered when the bound PVC gets deleted, causing the stored contents to be retained or automatically deleted.

Note PV and PVC provide a convenient way to decouple storage operations from workload deployment. The original aim was having storage administrators pre-provisioning PVs, allowing PVC to search the free ones and bind to the ones satisfying the claim as necessary.

Modern Kubernetes environments often use `Container Storage Interface` (`CSI`) plugins to automate the provisioning on the fly of volumes matching PVCs and manage their lifecycle.

In our example scenario, the `statefulset.yml` manifest does not contain the definition of any Persistent Volume.

Since on Killercoda the PV is automatically provisioned, it means they configured a `CSI` plugin, which automatically provisioned the PV as soon as the PVC was created.

Service

A Service is an abstraction that provides a stable endpoint for accessing backend `Pods`, which may change over time due to scaling or updates.

To better illustrate the concept, let's apply it to the `kubernetes-bootcamp` deployment.

Currently, the deployment is running a single replica Pod. This means that anybody willing to connect can directly connect to the Pod.

But what happens if we scale up the Pod, for example, by running

```
kubectl -n carcano scale deployment kubernetes-bootcamp \
--replicas=2
```

If you now check the Pods, by running

```
kubectl -n carcano get pods -l app=kubernetes-bootcamp
```

So, as we requested, they have now been scaled to two.

In my case I have

- kubernetes-bootcamp-9bc58d867-8w8wn
- kubernetes-bootcamp-9bc58d867-khcth

It should now be clear that we need a load-balancing service in front of them, to have a stable endpoint that clients can use to reach both Pods.

To achieve this, we create a `ClusterIP` service by running the following statement:

```
kubectl -n carcano expose deployment kubernetes-bootcamp \
--type=ClusterIP --port=80 --target-port=8080
```

We can get a human-readable description by running

```
kubectl -n carcano describe service kubernetes-bootcamp
```

The output is as follows:

```
Name:                     kubernetes-bootcamp
Namespace:                carcano
Labels:                   app=kubernetes-bootcamp
Annotations:              <none>
Selector:                 app=kubernetes-bootcamp
Type:                     ClusterIP
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.99.57.47
IPs:                      10.99.57.47
Port:                     <unset>  80/TCP
```

```
TargetPort:                8080/TCP
Endpoints:                 192.168.0.6:8080,192.168.0.7:8080
Session Affinity:          None
Internal Traffic Policy:   Cluster
Events:                    <none>
```

The output is very clear and self-explanatory: to summarize, it created a load balancer service on port 80 having IP `10.99.57.47` with `192.168.0.6:8080` and `192.168.0.7:8080` as backends.

Note ClusterIP is just one of the Service types available on Kubernetes.

Mind that, if readiness checks are configured for the Pods, then a `Service` object also monitors the actual health of each backend endpoint, removing them from the Service if they fail during the readiness check and automatically re-adding them when the issue is resolved.

Moreover, in some specific scenarios, `Services` can also be headless, meaning they do not serve as a stable endpoint but rather provide a convenient way to discover which `Pods` are currently members of the Service.

An example use case for headless `Services` is in cluster applications that need to be aware of their current members.

Note Services are quite a big topic: to better figure out how they work, I strongly suggest you check the official Kubernetes documentation at https://kubernetes.io/docs/concepts/services-networking/service.

Ingress Rule

An `Ingress Rule` defines how HTTP or HTTPS requests are routed to specific services based on matching criteria, such as the destination FQDN or path.

As an example, let's see how to create an Ingress Rule to redirect incoming requests for `kubernetes-bootcamp.carcano.corp` to the `kubernetes-bootcamp` service in the `carcano` namespace.

Just run

```
kubectl -n carcano create ingress kubernetes-bootcamp \
--rule="kubernetes-bootcamp.carcano.corp/*=kubernetes-bootcamp:80"
```

We can get a human-readable description of the Ingress Rule we have just set by running

```
kubectl -n carcano describe ingress kubernetes-bootcamp
```

The output is as follows:

```
Name:             kubernetes-bootcamp
Labels:           <none>
Namespace:        carcano
Address:
Ingress Class:    <none>
Default backend:  <default>
Rules:
  Host                              Path  Backends
  ----                              ----  --------
  kubernetes-bootcamp.carcano.corp
                                    /   kubernetes-bootcamp:80 (192.168.0.
6:8080,192.168.0.7:8080)
Annotations:                        <none>
Events:                             <none>
```

Be wary that these rules are part of the configuration for an Ingress resource, which is managed by an Ingress Controller.

The Ingress Controller is an application deployed on Kubernetes which acts as a reverse proxy to enable external HTTP or HTTPS access to services running within a Kubernetes cluster. Commonly used implementations are NGINX, HAProxy, or Traefik services.

Note Killercoda does not have any Ingress Controller set up by default, so the outcome of the above example, which is just provided to better illustrate the concept, cannot be tested.

Config Map

A `ConfigMap` is a Kubernetes resource that acts as a key-value store, where the values can range from simple scalars to entire documents, such as those loaded from files.

`ConfigMaps` allow you to separate configuration data from application code, making it easier to manage and update configurations without modifying the container images.

Since the `statefulset.yml` file contains also the definition of a configMap, we can view it by running

```
kubectl -n carcano get configmap
```

The output is as follows:

```
NAME               DATA   AGE
kube-root-ca.crt   1      42m
postgres-config    2      23m
```

Let's describe the `postgres-config` configMap:

```
kubectl -n carcano describe configmap postgres-config
```

As you can see, it provides two keys, `POSTGRES_DB` and `POSTGRES_USER`, along with their related values.

These keys are then injected into containers. If you look at the contents of the `statefulset.yml` file, they are passed as environment variables.

Anyway, they can also be presented as files as well, mounting them in the container's filesystem, allowing the value to be accessed by reading the mounted file.

Note The official documentation about `ConfigMaps` is available at https://kubernetes.io/docs/concepts/configuration/configmap.

Secret

A `Secret` in Kubernetes is conceptually like a `ConfigMap`, but it is specifically designed for storing sensitive data, such as passwords, certificates, and cryptographic keys.

While Secrets are `base64` encoded, it is important to note that this encoding is not encryption: it merely transforms the data into a different format. Therefore, anyone with access to the `Secret` can easily decode it.

The primary purpose of `Secrets` is to separate sensitive information from regular configuration data, promoting better security practices.

Since the `statefulset.yml` file contains also the definition of a Secret, we can view it by running

```
kubectl -n carcano get secret
```

the output is as follows:

```
NAME              TYPE     DATA   AGE
postgres-secret   Opaque   1      32m
```

Let's describe the `postgres-secret` Secret:

```
kubectl -n carcano describe secret postgres-secret
```

As you can see, it provides only the `POSTGRES_PASSWORD` key, along with its related value.

Just as with ConfigMaps, Secret's keys are then injected into containers. If you look at the contents of the `statefulset.yml` file, `POSTGRES_PASSWORD` is passed as an environment variable.

Anyway, mind also that Secret's keys can be presented as files as well, mounting them in the container's filesystem, allowing the value to be accessed by reading the mounted file.

Note The official documentation about Secrets is available at https://kubernetes.io/docs/concepts/configuration/secret.

Custom Resource Definitions and Operators

Kubernetes allows extending its set of Resource Definitions through `Custom Resource Definitions` (`CRDs`).

This capability enables users to enhance Kubernetes features by creating `Operators`. `Operators` are software extensions that automate the management of complex applications by encapsulating the operational knowledge required to manage them.

For example, there are `Operators` like `cert-manager`, which automatically generate and manage the lifecycle of certificates.

Additionally, there are `Operators` that implement the `Container Storage Interface` (`CSI`) for on-demand provisioning of `Persistent Volumes` as soon as unbound `Persistent Volume Claims` (`PVCs`) are detected.

This extensibility allows Kubernetes to adapt to various use cases and operational requirements effectively.

Another use case for `Operators` is managing clusters, for example, database clusters. An Operator can configure all the cluster-related settings for a specific database engine, automatically updating them when horizontal scaling occurs.

Note A proper discussion on Kubernetes would require a whole book, so we won't go any further. Anyway, my advice is to have a look at their very good documentation. A good place to start from is `https://kubernetes.io/docs/concepts`.

Helm

Note Since we still don't have a local Kubernetes instance, all the Helm labs in this chapter are supposed to be run on Killercoda using the Kubernetes scenario.

Kubernetes is an amazing orchestrator, but deploying and maintaining complex solutions on it is not as easy as it may look: a typical application deployed on Kubernetes consists of several dependent Resource Definitions for defining deployments, containers, volumes, services, Ingress Rules, load balancers, config maps, secrets, and so on.

These Resource Definitions themselves may contain a huge number of attributes defining every aspect of the application to deploy and how to orchestrate it, such as the required capacity for spawning up a Pod and the maximum granted capacity in terms of CPU and RAM, the service availability policy to meet during upgrades and so on.

The obvious outcome is that the process of managing complex Kubernetes applications is cumbersome and prone to human errors. To mitigate this complexity, they developed Helm.

Helm is an open-source package manager that enables developers and operators to define, install, and manage Kubernetes applications through a templated system of reusable packages known as `Charts`.

The primary outcome of using Helm is the streamlined deployment and update process of applications, which is automatically performed by generating the necessary Kubernetes Resource Definitions through the merging of charts' templates with deployment-specific values.

Additionally, Helm manages the upgrade process in accordance with defined service availability policies and facilitates quick and straightforward rollback procedures when necessary.

Helm charts follow versioning protocols and are published using Helm repositories, which can be hosted either through HTTP services or Git repositories.

The initial required step to utilize a specific Helm chart involves adding the Helm repository which is used to distribute it and then downloading the chart.

It's important to note that Helm automatically handles chart dependencies by downloading any required supporting charts that a primary chart may depend upon, ensuring complete application deployment without manual dependency resolution. This of course requires already having configured all the necessary Helm charts repositories.

Install Helm

Let's now have just a quick go with Helm: download the version of Helm suitable for your system from https://github.com/helm/helm/releases. For example:

```
curl -sL https://get.helm.sh/helm-v3.17.3-linux-amd64.tar.gz \
| tar xvz
```

Note If you are using Killercoda, mind that Helm is already installed.

Add Repositories

Helm does not come with preconfigured repositories – running the `helm repo list` statement produces an empty list indeed.

To practice a bit we can add the Enapter Helm repository – just run

```
helm repo add enapter https://enapter.github.io/charts
```

This statement adds a Helm repository pointing to https://enapter.github.io/charts and assigns it the name enapter.

Search Helm Charts

Once a Helm repository has been added, it is possible to list all the charts it provides by running helm search repo followed by the name of that repository. For example:

```
helm search repo enapter
```

The output is

```
NAME           CHART  VERSION APP VERSION DESCRIPTION
enapter/keydb 0.48.0 6.3.2               A Helm chart for
                                         KeyDB multimaster
```

The output reveals that this repository provides only the keydb Helm chart for deploying the KeyDB application.

Note KeyDB is an evolution of Redis that supports multi-master replication. At the time of writing this book, it is maintained by Snapchat.

Pull Helm Chart's Contents

Helm charts can be pulled as necessary.

Going on with the example, we can download the keydb chart, extracting it from the tarball archive used to package – just run

```
helm pull --untar enapter/keydb
```

This gives us the opportunity to have a closer look at a Helm chart: the contents are extracted into the keydb directory.

Let's have a look at its structure, checking the directory tree's contents:

```
find keydb
```

As anticipated, this is a very simple Helm chart, containing only the bare minimum objects:

- The `Chart.yaml` file, with the Helm chart's metadata
- The `templates` directory, containing the template files that are used to generate the deployed Kubernetes Resource Definitions
- The `values.yaml` file with the values that will be merged with the above templates – this file is generally called the values file
- The `Readme.md` file, used to document the `values.yaml` file and how to use the chart in general

---

Note I suggest you spend some time digging into each of these files, so as to better understand how Helm charts work.

Deploying Helm charts requires us to spend time reading the `Readme.md` file as well as inspecting the shipped `values.yaml`, figuring out how to change the values to fit our needs.

Anyway, mind that it is not unusual having to dig also in the templates looking for undocumented (or more often poorly documented) values.

Deploy Using Helm Charts

The best way to set up Helm-based deployments is organizing them into a tidy structure.

This approach enables procedural consistency, which provides several benefits, such as promoting scalability, simplifying operating the deployments, improving operational speed and reducing errors.

As an example structure, create the `deployments` directory:

```
mkdir deployments
```

We will use it to store the value files specific for every deployment.

Create Deployment-Specific Values Files

This is done by simply deriving a deployment-specific values file from the default values file (`values.yaml`) of the Helm chart which will be used.

The name of this deployment-specific values file must enable you to easily guess its purpose – for instance, it can be the chart's name followed by deployment specific identifiers, such as the instance number, the environment and security tier.

For example, assuming we are deploying the instance `0` in the testing environment security tier 1 (`t1`), we can copy the default values file from `keydb` the Helm chart into the `keydb-0-t1.yaml` file:

```
cp keydb/values.yaml deployments/keydb-0-t1.yaml
```

We can then adjust the `deployments/keydb-0-t1.yaml` as necessary.

In this specific scenario, to fit Killercoda lab's resources, we must deploy a single replica – just merge the below snippet with the current values in the file:

```
nodes: 1
multiMaster: "no"
```

Perform a Dry-Run

Before running the deployment statement, it's always best to run `helm template` to check how the generated Kubernetes Resource Definitions will look:

```
helm template -f deployments/keydb-0-t1.yaml keydb
```

As you can see, the path to the values file is specified by the `-f` option, whereas the last word is the name of the chart we want to deploy.

The rendered outcome will be printed to the standard output: this allows us to check it to ensure it looks as expected before running a real deployment.

Deploy the Instance

As per best practice, deployments are supposed to be provisioned within a dedicated namespace.

To meet this recommendation, we create on Kubernetes the `keydb-0-t1` namespace dedicated to the solution's instance we will deploy:

```
kubectl create namespace keydb-0-t1
```

Once the target namespace is created, we can now deploy the solution by running `helm install` as follows:

```
helm install --namespace keydb-0-t1 \
-f deployments/keydb-0-t1.yaml keydb keydb
```

In this command

- The `--namespace` option specifies the target namespace (e.g., `keydb-0-t1`)
- The `-f` flag points to the values file (e.g., `deployments/keydb-0-t1.yaml`)
- The first `keydb` is the name of the Helm release (used to identify this deployment instance)
- The second `keydb` is the name of the Helm chart being installed

The output is

```
NAME: keydb
LAST DEPLOYED: Mon Apr 14 09:42:11 2025
NAMESPACE: keydb-0-t1
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Helm created the Resource Definitions in the Kubernetes cluster: we can display them along with their state by running

```
kubectl -n keydb-0-t1 get all
```

You can now play a little bit with the deployed KeyDB.

When you are done, to conclude our very quick walkthrough, you can uninstall the solution by running

```
helm uninstall --namespace keydb-0-t1 keydb
```

Note In this book, we will focus on using Rancher, which integrates Helm and utilizes it behind the scenes to deploy applications. For this reason, we won't delve further into running the `helm` command-line tool. However, it's important to note that having knowledge of the `helm` command line is a valuable skill, and I strongly recommend checking out its official documentation at https://helm.sh/docs/intro/using_helm. I suggest you also learn how to write your own Helm charts: you can find additional resources on this topic at https://helm.sh/docs/topics/charts.

Rancher

Rancher is a comprehensive software stack designed for managing multiple Kubernetes clusters, whether they are deployed on public clouds or on-premises.

It aims to simplify and accelerate the management of deployments by addressing various operational and security challenges.

With Rancher, DevOps teams can efficiently operate several Kubernetes clusters from a centralized and user-friendly web interface, which offers functionality like that of the `kubectl` command-line utility.

Additionally, Rancher supports its own extensions to Helm charts, known as Rancher charts. These extensions allow Rancher to render charts into fully featured input forms with multiple tabs, text boxes, combo boxes, and options, making the deployment process more user-friendly and eliminating the need to manually edit Helm's values file.

Rancher is available as a free community version, licensed under the Apache License 2.0, which permits anybody to use, modify, and distribute the software freely.

For those interested in additional features and official support, there is an option to purchase a subscription from SUSE, which provides access to a set of commercial features and dedicated support services.

Deploy the Rancher Single Node Docker Container

Note In this section we illustrate the Single Node installation of Rancher – despite it not being supported in the production environment, it is a very convenient way to deploy Rancher for the first time and start playing with it.

In addition to that, migrating it to a production-ready Rancher on a dedicated Kubernetes cluster can be very easily done by using the backup operator, so this installation is the best and fastest option for starting to play with Rancher.

Before starting to deploy it, it is certainly worth the effort to spend some words describing the components of this solution as shown in Table 31-1.

Table 31-1. *Rancher single-node container components*

Component	Description
Containerd and Docker	Developed by Docker, `containerd` is a runtime using the Open Container Initiative (OCI) compliant runtimes built to run containers inside an abstraction layer, which manages namespaces, cgroups, union filesystems, networking capabilities, and more. It was donated in March 2017 to the Cloud Native Computing Foundation (CNCF). While `containerd` just manages the containers themselves, the Docker engine provides additional functionality such as Volumes, Networking and so on.
K3s	K3s is a fully certified by CNCF lightweight (around 50MB in size, requiring around 512M of RAM) Kubernetes distribution created by Rancher Labs: simply put, it is Kubernetes stripped out of non-essential components and using a different backing datastore (SQLite by default). This means K3s is not a fork, as it doesn't change any of the core Kubernetes functionalities and remains close to stock Kubernetes.
Rancher	Rancher is a Kubernetes management tool supporting multiple clusters not only deployed on premises, but also on public clouds. It centralizes authentication and role-based access control (RBAC) for all of the managed clusters and enables detailed monitoring and alerting for clusters and their resources, shipping logs to external providers. It also enables an easy workload deployment thanks to its direct integration with Helm via the Application Catalog.

Note When dealing with the Single Node Docker setup, Rancher is automatically deployed using a Helm chart on top of a K3s Kubernetes running in a Docker container.

Prerequisites

As we said, in a Single Node Docker setup, Rancher is deployed on K3s, which leverages a Docker-In-Docker-like setup. There are of course some prerequisites that must be configured before deploying it.

Note Since (sadly) Podman is not supported, if it is installed, we must uninstall it by running `sudo dnf remove -y podman buildah`.

As per best practices, the very first requisite is to update the platform:

```
sudo dnf update -y
```

If it is not already configured, add the Docker Community Edition DNF repository:

```
sudo dnf config-manager --add-repo \
https://download.docker.com/linux/rhel/docker-ce.repo
```

The K3s version provided by the Rancher container seems quite strict in terms of requirements: while writing the book I got stuck because of mismatches between the shipped K3s and the Docker version. I was able to address them only by installing specific packages' versions. When I was about to complete this book, they apparently fixed every problem.

If it happens again for your convenience this is the list of package versions I used on an Oracle Linux 9.6 x86_64:

- containerd.io-1.7.22-3.1.el9.x86_64
- docker-buildx-plugin-0.16.2-1.el9.x86_64
- docker-ce-cli-27.2.1-1.el9.x86_64
- docker-ce-rootless-extras-28.3.1-1.el9.x86_64
- docker-ce-27.2.1-1.el9.x86_64

- container-selinux-2.237.0-1.el9_6.noarch
- fuse-overlayfs-1.14-1.el9.x86_64
- libslirp-4.4.0-8.el9.x86_64
- slirp4netns-1.3.2-1.el9.x86_64

Because of these strict version requirements, if Docker is already installed in your system, remove it as follows:

```
sudo dnf remove -y docker-ce docker-ce-cli \
containerd.io docker-buildx-plugin docker-compose-plugin
```

then, install the Docker RPM packages specifying also the desired version:

```
sudo dnf install -y containerd.io-1.7.22 \
docker-buildx-plugin-0.16.2 docker-ce-cli-27.2.1 \
docker-ce-rootless-extras docker-ce-27.2.1
```

To prevent undesired side effects when updating, let's lock the most critical RPM packages.

First, install the DNF versionlock plugin:

```
sudo dnf install -y 'dnf-command(versionlock)'
```

Then, proceed with the version locking as follows:

```
sudo dnf versionlock add containerd.io docker-buildx-plugin \
docker-ce-cli docker-ce-rootless-extras docker-ce \
container-selinux fuse-overlayfs libslirp slirp4netns
```

We must now of course enable the `docker` service to start at boot:

```
sudo systemctl enable docker
```

K3s mandatorily requires to load the following networking-related kernel modules:

- ip_tables
- ip_conntrack
- iptables_filter
- ipt_state

Just create the /etc/modules-load.d/rancher.conf file with the following contents, so to have them loaded at boot:

```
ip_tables
ip_conntrack
iptable_filter
ipt_state
```

It is also mandatory to set the following Kernel tunables:

- net.bridge.bridge-nf-call-iptables
- net.bridge.bridge-nf-call-ip6tables
- net.ipv4.ip_forward

Just create the /etc/sysctl.d/rancher.conf file with the following contents, so to have them set at boot:

```
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
```

Since Rancher will expose ports 80 for http and 443 for https traffic, if you are running firewalld, you must also add the related firewall exceptions:

```
sudo firewall-cmd --permanent --add-service=http
sudo firewall-cmd --permanent --add-service=https
```

We must now restart the system to have all the changes we did so far applied:

```
sudo shutdown -r now
```

We have finally reached the stage of the actual Rancher's deployment.

Deploy Rancher

Rancher consists of several containers running inside the K3s: since containers are disposable objects by design, we must create the directory tree to be used as the mount point for persisting the data.

- /var/lib/rancher: All the Rancher data

- /var/log/rancher: Rancher's log files
- /var/log/rancher/auditlog: Rancher's audit log files

Create them as follows:

```
sudo mkdir -m 755 /var/lib/rancher /var/log/rancher \
/var/log/rancher/auditlog
```

We are now ready to create and run the Rancher's K3s container – just run

```
sudo docker run --name k3s -d --restart=unless-stopped \
--privileged -e AUDIT_LEVEL=1 -p 80:80 -p 443:443 \
-p 30000-31000:30000-31000 \
-v /etc/localtime:/etc/localtime:ro \
-v /dev/urandom:/dev/random:ro \
-v /var/log/rancher/auditlog:/var/log/auditlog \
-v /var/lib/rancher:/var/lib/rancher \
rancher/rancher:v2.10.3
```

The above statements create and start from the rancher/rancher:v2.10.3 image a detached (-d option) container caller k3s with the following options:

- Runs in privileged mode.
- Gets automatically restarted unless it was stopped on purpose.
- The host's ports 80 and 443 are mapped to the same ports inside the container.
- The port range 30000-31000 is mapped to the same port range inside the container – we will use if for exposing workload running on Kubernetes using Node Port services as necessary.
- Bind mounts /etc/localtime the timezone file (the timezone file) and /dev/random (entropy source) in read-only mode.
- Bind mounts /var/lib/rancher to provide data persistence.
- Bind mounts /var/log/rancher/auditlog to persist log files.
- Sets the AUDIT_LEVEL environment variable to 1.

For the sake of completeness, mind that the deployed rancher version is always the same as the container image (`2.10.3`).

Depending on the system and network bandwidth, this operation takes some time.

Since K3s prints its logs into the `k3s` container's standard output, we can easily monitor the Rancher's deployment process by typing

```
sudo docker logs -f k3s 2>&1
```

Note To detach from the log, just type `CTRL+C`.

Operating K3s

It is certainly worth the effort to spend a few words on how to operate K3s to be able to jump into it as necessary.

Simply launch a bash shell inside the `k3s` container as follows:

```
sudo docker exec -ti k3s bash
```

Once done, we can operate as with any other Kubernetes cluster using the `kubectl` command-line utility.

For example, to display the cluster's status, run

```
kubectl get nodes
```

The output must look like

```
NAME       STATUS ROLES                       AGE   VERSION
local-node Ready  control-plane,etcd,master 5m37s v1.31.1+k3s1
```

Exit from the `k3s` container by typing `exit` or pressing `CTRL+D`.

Having to jump inside the `k3s` container each time for running `kubectl` statements can quickly become annoying: we can add an alias to emulate running the `kubectl` statement directly from the shell.

Simply create the `/etc/profile.d/k3s.sh` file with the following contents:

```
alias kubectl="sudo docker exec -ti k3s kubectl"
```

After disconnecting and reconnecting to have the alias applied, we must now be able to run the `kubectl` statements directly from the shell.

For example, let's have a look at the whole workload deployed on the K3s Kubernetes:

```
kubectl get all --all-namespaces
```

Connection to Rancher

In a lab environment, using a certificate from a global Certificate Authority (CA) is unnecessary and cumbersome.

To simplify lab's setups, Rancher automatically generates during the deployment a CA certificate and uses it to issue its own TLS server certificate.

Trust Rancher's CA Certificate

To avoid browser warnings, we must add Rancher's generated CA certificate to our browser's truststore.

Start by retrieving the certificates using the `retrieve-certs.sh` script from the 23rd chapter. Run the script as follows:

```
cd /tmp
~/devsecops-pillars/ch23/retrieve-certs.sh 127.0.0.1
```

This will save both the CA and server certificates in the `/tmp` directory.

Next, identify the CA certificate, which is the file starting with `dynamiclistener-ca@`, such as `dynamiclistener-ca@1743709217.crt`.

Unless you are using the web browser in the VM itself (which is unlikely), you now need to copy this file to the machine where you will use the browser.

If you have provisioned the lab VM using the `Vagrantfile` provided with the book, the easiest option is to move the file to the `/vagrant` directory, which is shared with your host. If shared folders don't work, or you are not using Vagrant, you can fetch that file by running an SCP statement on your host. For example:

```
vagrant scp devsecops_lab_01:/tmp/dynamiclistener-ca@*.crt.
```

As a last resort, since the file to copy is PEM-encoded text, you can just print its contents to the terminal using `cat /tmp/dynamiclistener-ca@*.crt`, then copy and paste the output into a text editor on your host.

Once you have the CA certificate, import it into your browser's truststore.

Note The exact steps for trusting the certificate depend on your browser – if you don't know how to proceed, search for instructions like "[browser name] add trusted certificate." Make sure to add it to the CA certificates section and enable any trust-related options. Remember, after your lab experiments, remove this certificate to avoid security risks.

Map Rancher's FQDN to the Host's File

In addition to trusting the certificate, there is another mandatory step:

Rancher's TLS certificate is valid for the fully qualified domain name (FQDN) `rancher.cattle-system`, which is a unique identifier for the domain. To access it, map this name to your lab VM's IP address in your machine host's file.

Note To find the IP address of your lab's virtual machine (VM), run the command `ip -4 a` in the VM's terminal. This will list the IPv4 addresses associated with its network interfaces. If you launched the VM using the `Vagrantfile` provided with the book, the relevant IP address is typically assigned to the `eth1` interface. If you're unsure which IP to use, check for the one that shares the same subnet as your host machine's IP address. You can verify this by comparing network prefixes (in my own scenario, both starting with `192.168.66.x`).

Once you've got your lab's VM IP address, edit the host's file on your machine. On Linux or macOS, modify `/etc/hosts`.

On Windows, edit `C:\Windows\System32\drivers\etc\hosts`.

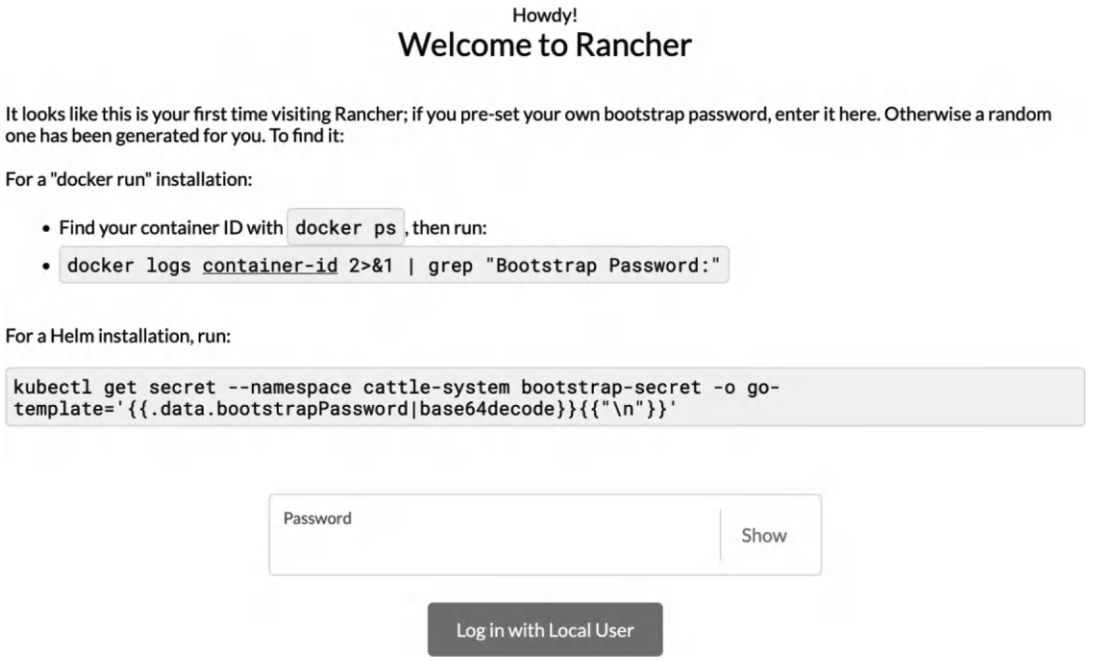
Add or update a line with your VM's IP address followed by `rancher.cattle-system`, for example:

```
192.168.66.177 rancher.cattle-system
```

Complete the Welcome Procedure

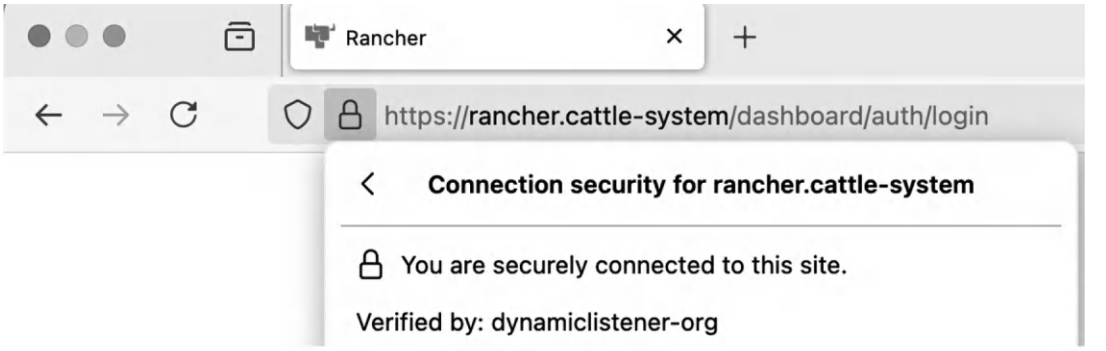
Once you've trusted the certificate and updated the host's file, open `https://rancher.cattle-system` in your browser.

You should now be able to access Rancher without warnings, showing you the following welcome page:



If instead you see security warnings, it means you have not properly trusted the CA certificate: in such a case, you must fix it as necessary.

For your reference, the following graph shows how the TLS server certificate tooltip looks on Firefox when the CA certificate is properly trusted:



564

As suggested by the hint in the Rancher's welcome page, grab the automatically generated one-time administrative password by running

```
docker logs k3s 2>&1 | grep "Bootstrap Password"
```

Enter it in the password textbox, specify you want to provide the password yourself and enter the one you want to assign to the `admin` user. Remember also to tick the option to accept the End User License Agreement.





Once done, complete the first login.

Note If you need to shut down the lab virtual machine, you must first stop the `k3s` container running Rancher by typing `docker stop k3s`, or corruptions may happen. After this, remember also to stop the `Pulp` Docker deployment as explained in the previous chapter. At system boot, the `Pulp` Docker deployment will automatically start, whereas you must manually start the `k3s` container running Rancher by typing `docker start k3s`.

Rancher Backups

Once Rancher is installed, the very first thing to set up is scheduling Rancher's backups so as to be able to quickly restore if anything bad would happen. For this purpose, Rancher provides the `Rancher Backups` app.

Prepare the Backing Space

In a real-world environment, the backup application will store backups into an S3 bucket or a pre-provisioned or provisioned on the fly Persistent Volume (PV) backed by a NAS or a SAN automatically bound to its Persistent Volume Claim (PVC).

Since this is a Lab, and this Rancher is deployed on a containerized K3s Kubernetes, to keep hardware requirements minimal and simplify the process, we use the Docker host's filesystem as the backend device of this PV.

Since the `k3s` container has the `/var/lib/rancher` bind-mounted to the Docker host's filesystem to provide persistence, we can simply create the `/var/lib/rancher/backups` subdirectory and use it as the backup's backing space.

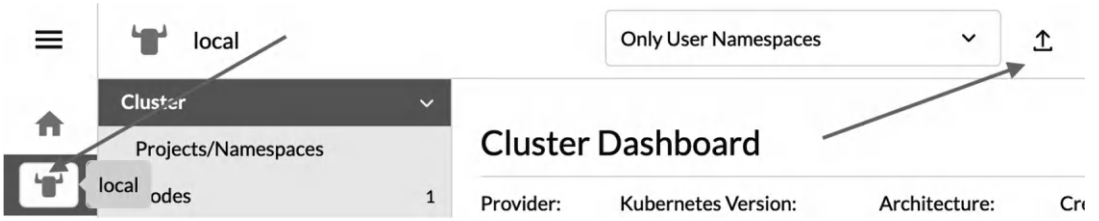
On the Docker host (the lab VM), just run

```
sudo mkdir -m 750 /var/lib/rancher/backups
```

Provision the Persistent Volume

We are now ready to define the Persistent Volume which will use the backing space we just created.

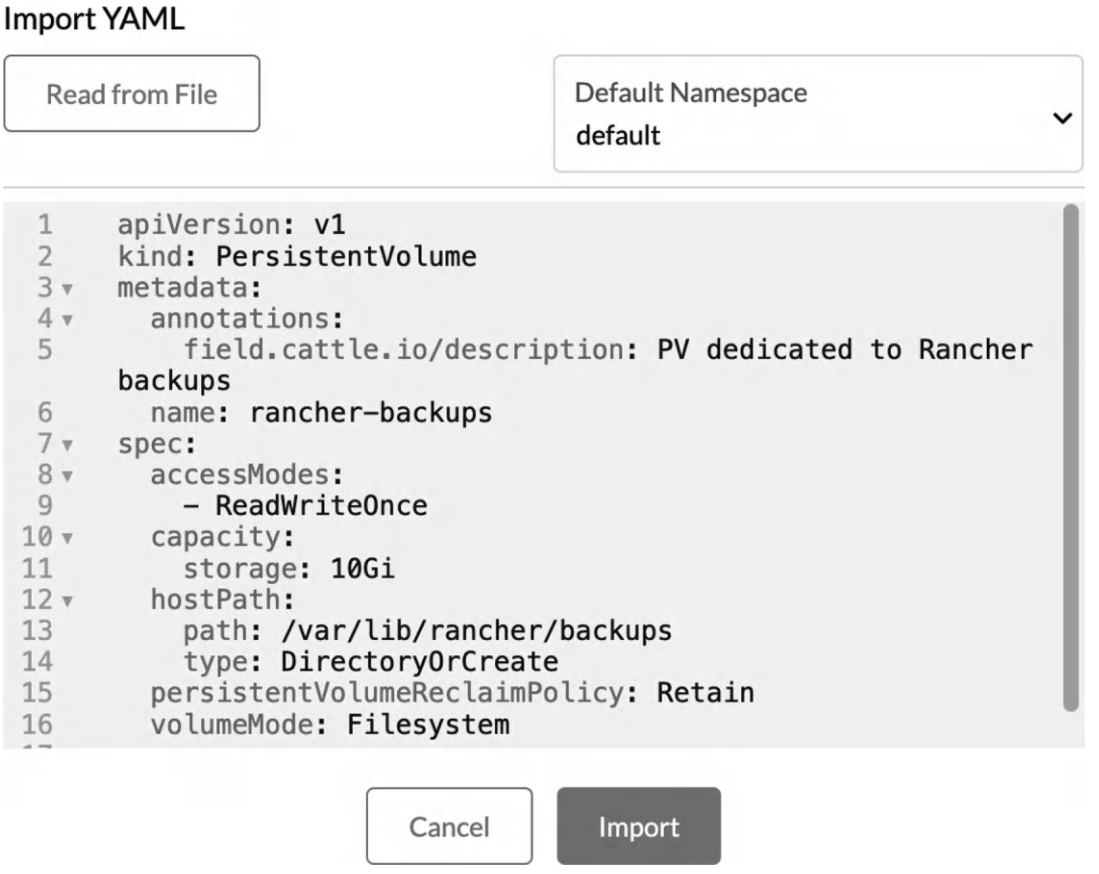
Within the Rancher's web UI, select the `local` cluster (the button with the bull's head) and click on the `Import YAML` button:



This chapter's workspace directory contains the `pv.yml` file with the Resource Definition of the PV to bind the `Rancher Backups` app.

This PV, called `rancher-backups`, is configured to bind mount to the `/var/lib/rancher/backups` on the lab's VM directory.

Open the `pv.yml` file and paste its contents in the textbox as shown by the below graphic:

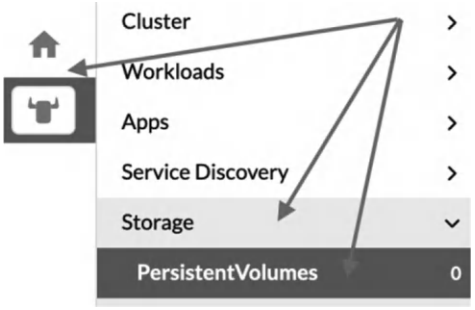


The only setting you may want to change is the storage capacity: leave all the rest as they are.

Once done, just click on the `Import` button: the `rancher-backups` Persistent Volume will be immediately provisioned.

Note We must of course then implement a backup procedure from the Docker host's filesystem, to avoid losing the data if the Docker host breaks. Since this is highly dependent on your scenario, it is up to you to implement it.

Let's have a look at the just-created Persistent Volume (PV).



Within the Rancher's web UI, select the `local` cluster (the button with the bull's head), expand the `Storage` group and click on `PersistentVolumes`: this displays the list of available Persistent Volumes as shown by the below graphic:

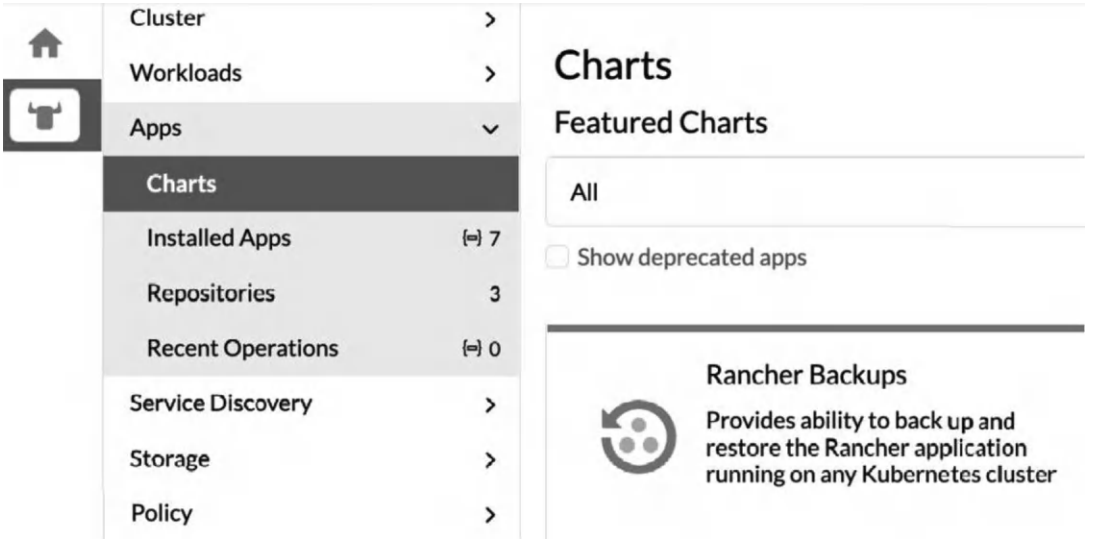
State	Name	Reclaim Policy	Persistent Volume Claim	Source	Reason	Age
Available	rancher-backups	Retain	n/a	HostPath		39 secs

Provision the Backups App

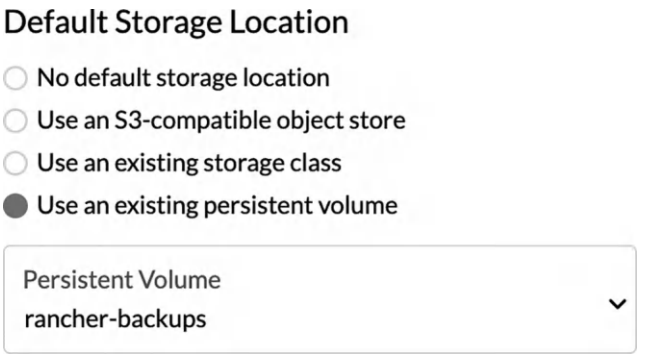
We are now ready to deploy the `Rancher Backups` application: since its Rancher chart is available, we can easily deploy it from the Rancher's Web UI.

Within the Rancher's web UI, select the `local` cluster (the button with the bull's head), expand the `Apps` group and click on `Charts`: this will display all the available charts.

Using the filter textbox, type `Rancher Backups` so to get only the Rancher Backups application: click on it to display its `Readme` page, then deploy it using the `Install` button on the top right.



From the `Install into project` combo box, select the `System` project and click on the `Next` button. The following form will be shown



From the `Default Storage Location` list, tick `Use and existing persistent volume` and then select the `rancher-backups` Persistent Volume we just created: this way the `Rancher Backups` PVC will be tightly bound to the `rancher-backups` PV we just provisioned.

Lastly, click on the `Install` button to deploy the `Rancher Backups` application.

Note As you can see, we didn't need to configure the values file – that is because this is a Rancher chart which, as we said, improves the Helm chart by rendering a configuration form.

Once the deployment completes, its containers will be available in the `cattle-resources-system` namespace.

Let's have a look at that namespace's contents. In the lab's VM terminal, run

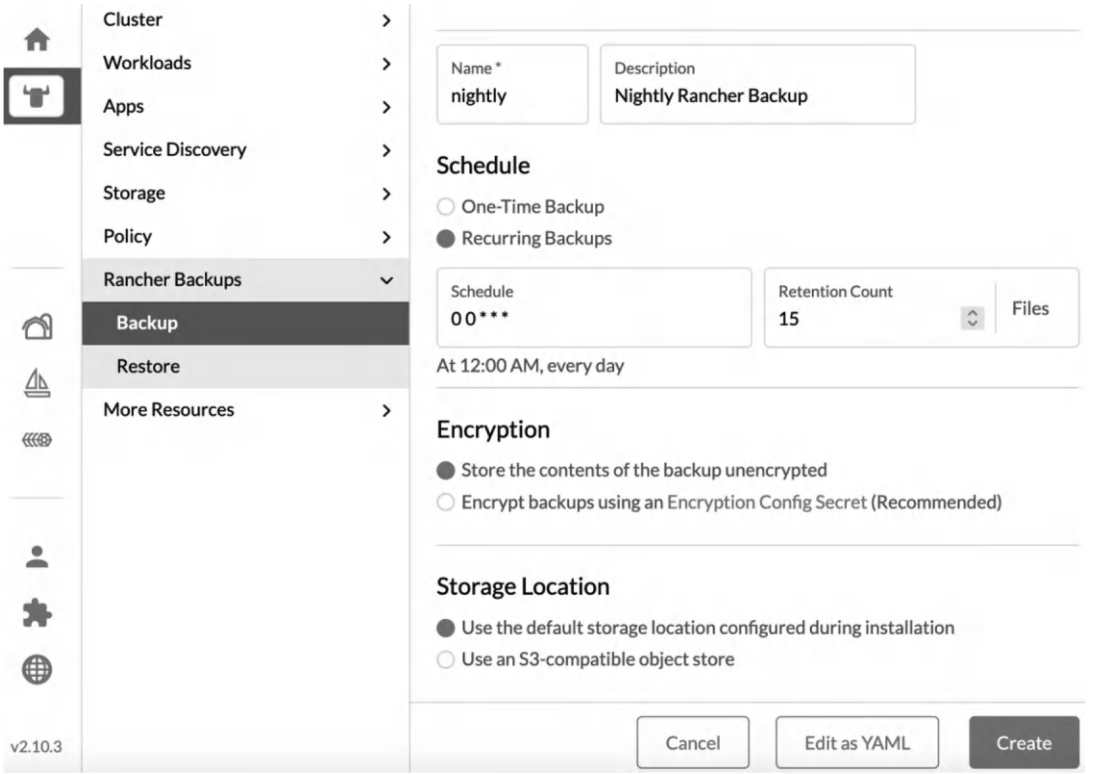
```
kubectl -n cattle-resources-system get all
```

Note I am not explaining the procedure for recovering since it would be a duplication of the official manual and it may be subject to changes.

Schedule Backups

We are still missing the last thing: scheduling a backup – select the `local` cluster (the button with the bull's head), expand the `Rancher Backups` group and click on `Backups`: this will display all the available backups.

Click on the `Create` button and fill in the form as shown below:



In this example we have just scheduled an unencrypted nightly backup with a 15-day retention.

Note Rancher is an amazing piece of software that can really ease your life when dealing with Kubernetes: I strongly suggest you read their official documentation at https://rancher manager.docs.rancher.com.

Active Directory Authentication

Rancher supports several external authentication sources: here we see how to configure it using Microsoft Active Directory.

Note Microsoft Active Directory is a service that integrates Kerberos, LDAP and RPC. It uses Kerberos for authentication and authorization, with Microsoft

extending the standard Kerberos protocol through features like the Privilege Attribute Certificate (PAC), which provides additional authorization information. Active Directory also implements LDAP to store accounts, configurations, and security objects, allowing it to function as an LDAP server and enabling applications and services to authenticate users via LDAP. Additionally, RPC is used for communication between domain controllers and clients, as well as for certain administrative tasks.

Before continuing with the read, mind that this section is not a practical lab, since trying this requires a running Microsoft Active Directory instance. It is just a reference I added for your convenience to provide you with an easy track you can check.

To configure the external authentication source

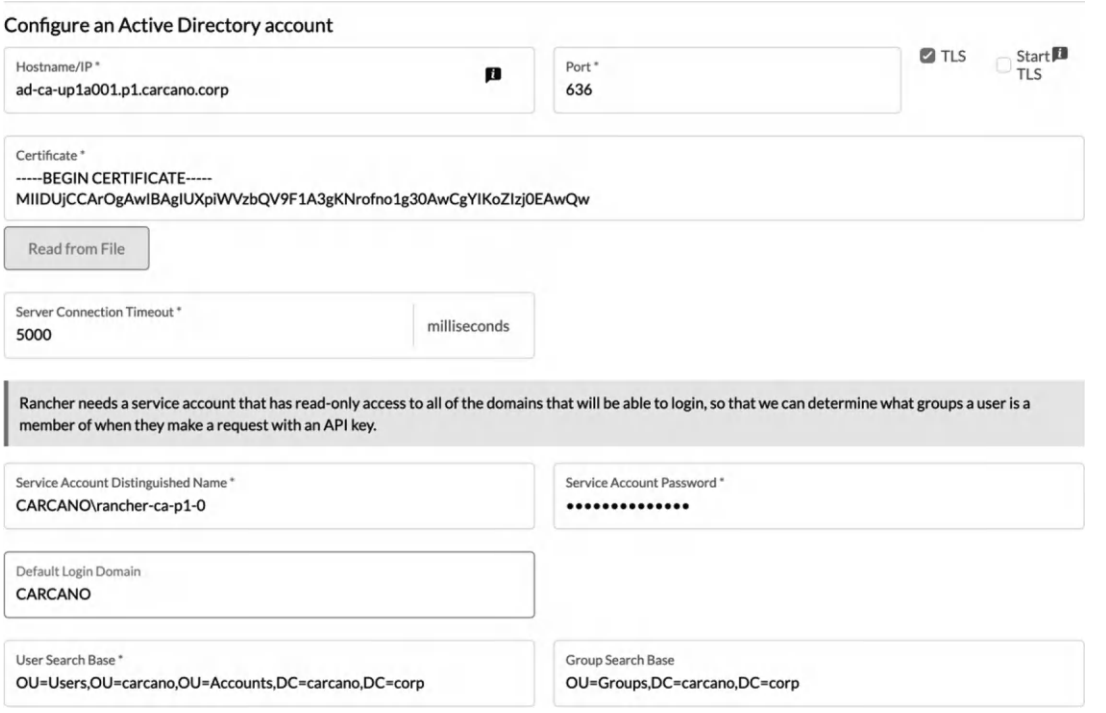
- Log into the Rancher UI using the initial local `admin` account.
- From the global view, navigate to `Users & Authentication`, then to `Auth Provider`.

Select and configure `ActiveDirectory` to load the related configuration form. Fill in the form as follows:

- Enter in the Hostname/IP the FQDN of the Active Directory service.
- If your Active Directory is configured to support LDAPS (as in the graphic below), tick the `TLS` option: the port is automatically switched to `636`. After this, by using the `Read from File` button, upload the Certificate Authority's certificate used to sign the Active Directory's service Certificate.
- Enter the `Service Account Distinguished Name`. Here the definition Distinguished Name can be misled – they are referring to the username, not to the account's LDAP DN.
- Here you must specify the username of the account used to bind the Active Directory Service, along with its password in the `Service Account Password` text box.

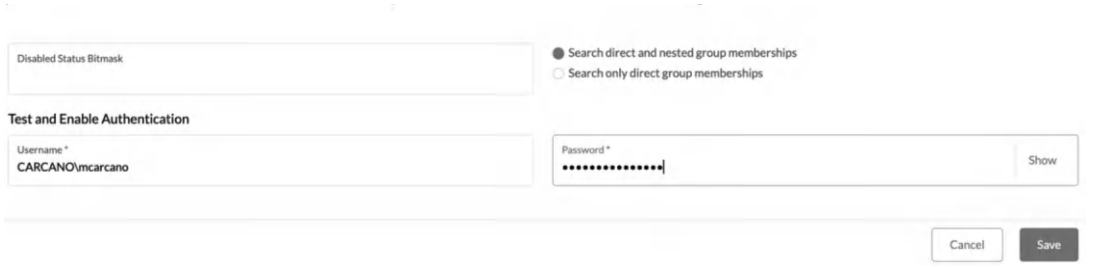
- Please note also that you must prepend the Active Directory REALM to the username. In this example the Active Directory realm is `CARCANO`, so the `Service Account Distinguished Name` is `CARCANO\rancher-ca-p1-0`.
- Enter the Active Directory realm in the `Default Login Domain` – in this example it is `CARCANO`: this is used as a default entry domain when users do not specify their user's realm when logging in to the Rancher web UI.

Enter the `User Search Base` and `Group Search Base` as in the below graphic: of course, modify the values according to your Active Directory setup.



Note If you want to use Active Directory nested groups in Rancher, remember to tick the `Search direct and nested group membership` option like in the below graphic.

At the very bottom of the form, enter a username and a password that you want to use to perform an authentication attempt to test the above settings.



Once done, just click on the Save button.

If everything is properly set, the authentication provider gets set, and it starts retrieving users and groups from the Active Directory.

The current setup enables any Active Directory users to log in to Rancher – the best practice is instead to have an LDAP group dedicated to each Rancher's instance, such as `Rancher-O`, and use its membership as the criteria for authorizing access to that Rancher instance.

To configure it, select the `Restrict access to only authorized users & groups` option and select the `Rancher-O` group as in the below picture.



Once done, disconnect the local `admin` user and login again with a corporate user: if it is a member of the `Rancher-O` Active Directory group, it must be granted access; otherwise the login will fail.

Note This just configures authentication to access the Rancher instance: it is then possible to tailor fine-grained access rules by binding specific Active Directory groups to specific Rancher roles on specific objects.

Summary

In this chapter, we explored key topics in the Kubernetes ecosystem, starting with an overview of popular solutions and diving into Kubernetes and Helm through practical labs for hands-on experience. We then focused on Rancher, demonstrating its installation as a single-node Docker container on K3s.

These concepts provide the knowledge and skills needed to operate confidently in cloud-native environments like Killercoda and on-premises setups. For DevOps and DevSecOps professionals, proficiency in Kubernetes, Helm, and Rancher is essential, given the growing reliance on container orchestration and microservices in modern software development.

Helm acts as a package manager for Kubernetes, simplifying application deployment by managing dependencies and configurations to ensure consistency across environments. Rancher complements this by offering a centralized platform for managing multiple Kubernetes clusters, whether on-premises or in the cloud.

Together, these tools enable DevOps and DevSecOps teams to foster better collaboration, speed up delivery cycles, and strengthen application security. In today's fast-paced world of rapid deployment and CI/CD, mastering Kubernetes, Helm, and Rancher is crucial for achieving successful outcomes in software development and operations.

PART XII

CI/CD Toolchain

Within modern systems engineering, automated delivery constitutes a baseline operational standard; DevOps and DevSecOps practitioners must possess the capacity to construct and govern a deterministic and resilient CI/CD toolchain. Advanced infrastructure management replaces manual, risk-prone deployments with event-driven execution models that safeguard both release consistency and business continuity. Driven by escalating public cloud operational expenditures and strict data sovereignty regulations, contemporary engineers must architect these automation engines within localized, self-hosted environments to maintain absolute content control. This section delivers the precise blueprint for on-premises continuous integration tailored for Cloud Repatriation strategies. By deploying Gitea alongside dedicated, self-hosted actions runners, you will implement an event-driven automation layer that unifies version control, automated testing, and secure packaging into a cohesive local pipeline. Crucially, this setup integrates Pulp 3 as the centralized repository manager to govern localized artifact distribution, enabling you to orchestrate secure, cost-effective deployment cycles that systematically eliminate external cloud dependencies and operational liabilities.

CHAPTER 32

Deploying Gitea Using the Helm Chart

This chapter provides a hands-on opportunity to practice deploying workloads on Rancher, demonstrating how it simplifies the management of Kubernetes clusters for efficient application orchestration and scaling.

To explore these concepts, you'll deploy Gitea – a widely used CI/CD solution and a prerequisite for labs in subsequent chapters.

The lab guides you through the full deployment process, covering infrastructure requirements such as setting up a database and filesystem storage, Kubernetes essentials like preparing necessary secrets, and Rancher-specific steps including adding third-party Helm repositories.

At the chapter's end, you'll get a brief walkthrough of Gitea, where you'll create an organization, set up a Git repository, and configure and run an Actions workflow. This illustrates practical applications of CI/CD tools, which are crucial for automating software development processes.

The chapter emphasizes the value of using off-the-shelf Helm charts and third-party Helm repositories. These allow professionals to leverage community-driven resources, speeding up deployments and providing access to a broad array of pre-configured applications.

By configuring Actions workflows in Gitea, you'll gain experience that prepares you for real-world DevOps and DevSecOps scenarios, fostering streamlined and secure development practices.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_32

What Is Gitea

Gitea is an all-in-one solution not for just self-hosting the Git service: it is a full-featured SCM with a convenient web UI for code reviews and team collaboration, providing also CI features through Actions workflows.

With a design clearly inspired by GitHub, it is a good and very popular alternative to other Git products and hosting services.

In this book we are using the community available version, but if you fancy you can purchase support subscriptions as well as use the SaaS version called Gitea Cloud.

Note Before going any further, change to this **chapter's workspace directory** by running `cd ~/devsecops-pillars/ch32`.

Deploy Gitea Using the Official HELM Chart

In our example scenario, we are about to deploy the Gitea Testing 01 instance. For our convenience, we assigned it the label `gitea_t01`, so to have a short and meaningful identifier we can use each time it is needed.

Note This is a purely didactical lab aimed at keeping system requirements at bare minimum, so to be able to run our whole labs using just one Virtual Machine.

To achieve this, the lab completely lacks security and reliability, so don't use this setup in real life: it is just a playground to give you the opportunity to play, gaining practical experience so that you can manage real life scenarios on your own.

Requisites

Gitea requires both a filesystem for persisting the Git repositories as well as a database where to store its metadata.

PostgreSQL Database

Since we already provisioned a PostgreSQL instance for the Pulp3 deployment, to save resources, we reuse it also as Gitea's database backend: this way we just need to create the database used by Gitea and the username to access it.

Note If you deleted the Pulp 3 deployment, get back to Chapter 29 and deploy it again. Before undergoing a redeployment process, make sure the containers are not just stopped (`docker ps -a`) and, if they are, just start them using `docker start`.

Let's create Gitea's database and user account by running a few shell commands.

To be able to easily refer to it in the next statements, we need to get PostgreSQL's container ID and store it in the `PGSQL` variable. Just run

```
PGSQL=$(docker ps | grep postgres | awk '{print $1}')
```

We can now use it to inject `psql` statements in the container.

First, let's create the `gitea_t01` PostgreSQL user with password `grimoire` as follows:

```
docker exec ${PGSQL} psql -U pulp -c \
"CREATE USER gitea_t01 WITH PASSWORD 'grimoire';"
```

We can now create the `gitea_t01` database, assigning the `gitea_t01` user as owner:

```
docker exec ${PGSQL} psql -U pulp -c \
"CREATE DATABASE gitea_t01 WITH OWNER gitea_t01";
```

Filesystem Storage

On Kubernetes, filesystem storage is provisioned through Persistent Volumes (PVs).

In this small lab environment, without CSI automations to automatically provision PVs in response to Persistent Volume Claims (PVCs) during deployment, we must manually pre-provision these PVs.

To keep requirements minimal, we'll use the Docker host's filesystem as the backend device for these PVs. Specifically, we need to create two PVs: one for the Gitea instance itself and another for the Gitea Actions Runner workflow.

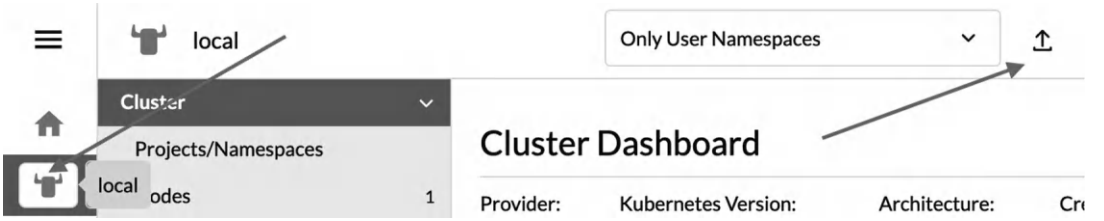
To do this, start by creating the backing directory trees on the Docker host where these PVs will be bound:

```
sudo mkdir -m 777 /var/lib/rancher/{gitea,gitea-act-runner-0}
```

Note For your info, `/var/lib/rancher/gitea` is used by Gitea to store its data, whereas `/var/lib/rancher/gitea-act-runner-0` is used by Gitea Actions Runner workflows.

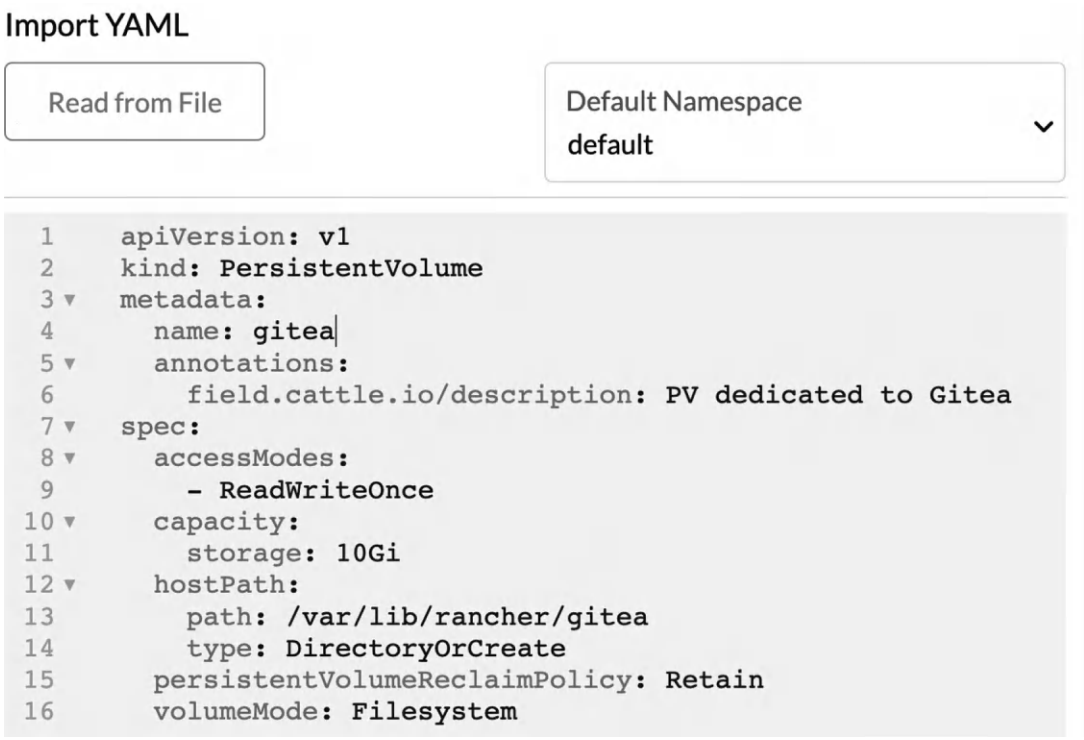
Now that we have the two backing store directory trees, we can move to the Rancher's web UI and create the PVs.

Select the `local` cluster (the button with the bull's head) and click on the `Import YAML` button:



The `gitea-pvs.yml` file in the chapter's workspace directory contains the Resource Definition for creating both PVs.

Open it and paste it in the textbox as shown by the below picture:



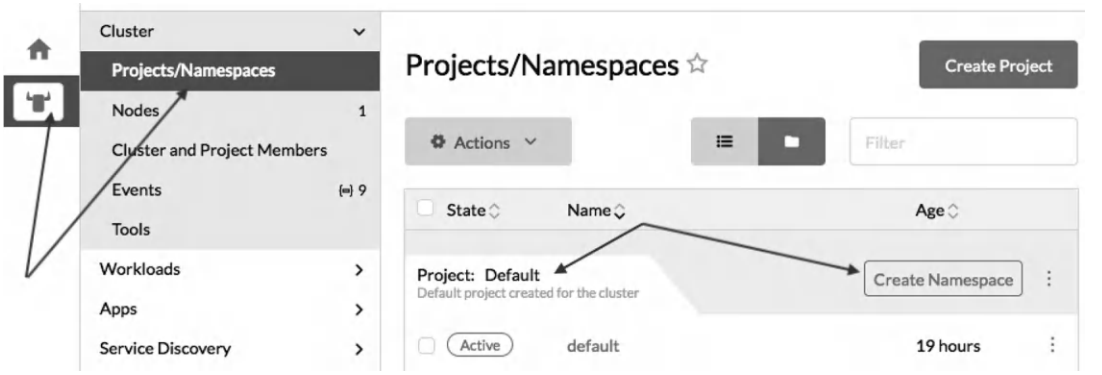
Once done, just click on the `Import` button: the two Persistent Volumes will be immediately provisioned and marked as `Available`.

Create a Namespace

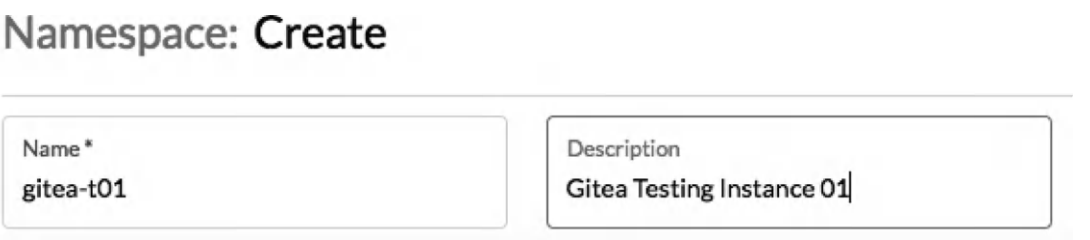
Since the best practice is to group deployments on a per-instance basis using namespaces, we now need to create the namespace where to provision Gitea.

Select the `local` cluster (the button with the bull's head) and from the left menu click on `Projects/Namespaces` within the group `Cluster`.

Then, click on the `Create Namespace` button belonging to the `Default` project.



Create the `gitea-t01` namespace, filling in the form as by the below picture:



Note In a real-life scenario, we would have assigned this namespace to purpose-related Project, such as `cicd`, creating it as necessary, to be able to tailor resources and access permissions as necessary.

Before continuing, select the `gitea-t01` namespace in the namespace filter combo box on the top right of the page: by doing this any further change will always apply to the `gitea-t01` namespace.

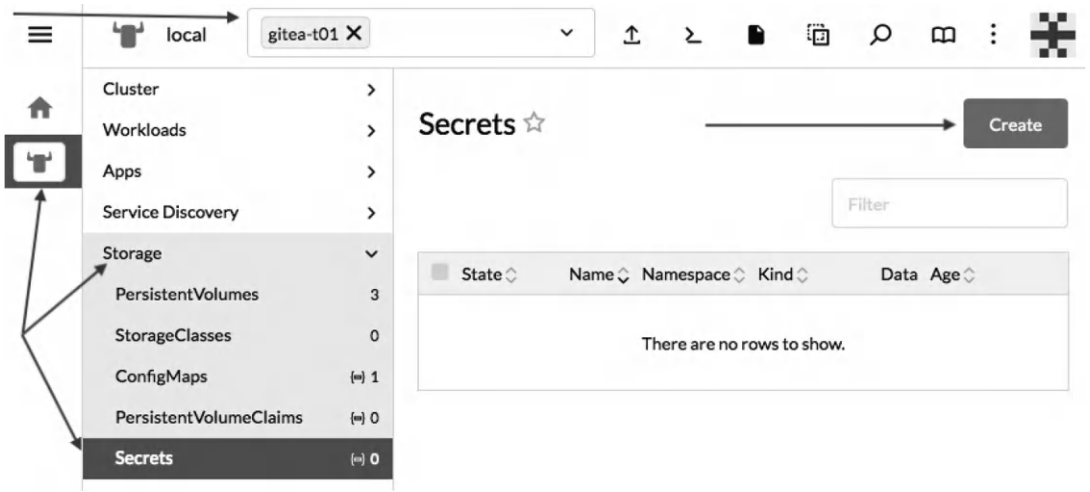
Note Namespaces are a convenient way for grouping workloads: in my experience a good naming scheme must be meaningful to let you guess which is the workload and at the same time suggest to you the environment and instance number.

Create the Administrative Credentials' Secret

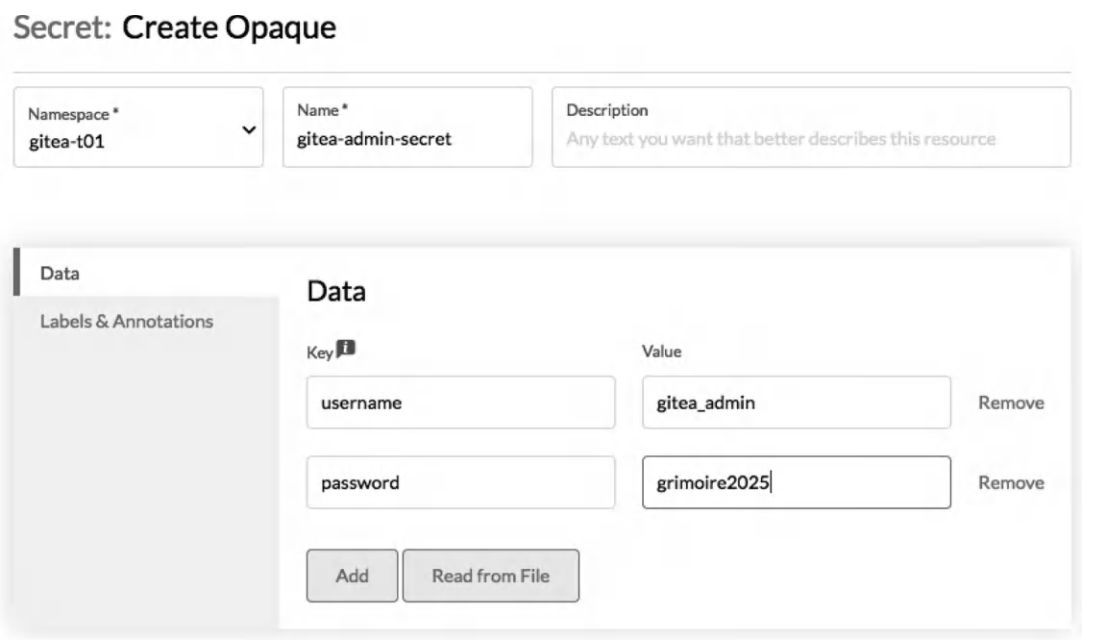
We must now create the secret for storing Gitea's administrative credentials, which are the ones you will use to log in to Gitea's web UI.

Select the `local` cluster (the button with the bull's head) and from the left menu expand the `Storage` group and click on `Secrets`.

Once done, click on the `Create` button.



This opens a page with multiple boxes, one for each type of secret: select the `Opaque` type secrets one, and fill in the form as shown in the following picture:



Note You can of course specify any value you want as password.

When done, click on the `Create` button on the bottom right of the page.

This way, we created in the `gitea-t01` namespace the `gitea-admin-secret` secret with the `username` and `password` key/values.

This secret will then be used by Gitea as the source for the administrative user's credentials.

Gitea Helm Charts

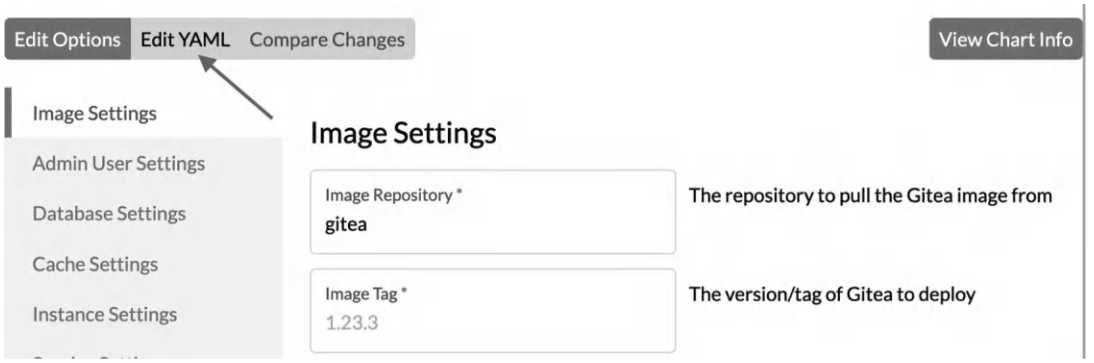
It is possible to deploy Gitea both by using the Rancher Partners' chart repository or the official Gitea Helm repository.

Gitea from the Rancher Partners' Chart Repository

The Rancher Partners' chart repository contains a downstream of the official Gitea Helm chart, improved to become a Rancher chart.

This is an enhanced version of Helm charts that includes an additional YAML manifest (`questions.yaml`), which renders visual controls (such as text boxes and check boxes) to reduce the effort required for manually editing the Helm chart values file.

The rendered questions file generates a form which looks like the one below:



The form is used to pre-fill the Helm chart's values file: if necessary, it is possible to quickly switch to directly edit the values file by clicking on the `Edit YAML` tab.

It is worth mentioning that Gitea Actions Runners were originally integrated into the main chart; the Gitea maintainers transitioned to a decoupled architecture to allow independent scaling. This was finalized in Gitea Helm Chart `v12.0.0` (released in 2025), which explicitly removed all remaining Actions Runner workflow-related templates. These are now maintained as a separate, standalone Helm chart.

This change had an impact on the Rancher chart from the Rancher Partners' chart repository: since they have not yet ported the new Gitea Actions Runner workflow Helm chart into it, it is no longer possible to deploy Actions Runner workflows using the most current chart's version.

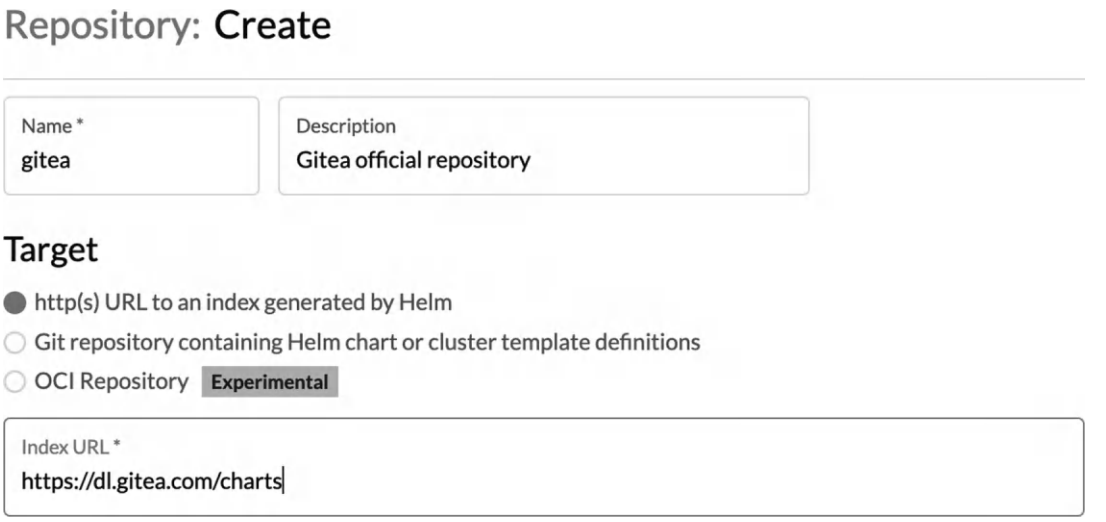
Gitea from the Official Chart Repository

At the time of writing this book, the only available way for deploying both Gitea and the Gitea Actions Runner workflows is configuring the official Gitea Helm chart repository, which provides both charts.

The downside of this method is that, since the provided charts are just Helm charts, they don't provide the `questions.yaml` file which is rendered in the Rancher's web UI to provide a more user friendly experience. That means that the installation must be conducted only relying on the Helm chart's values files.

To add the net Gitea's official Helm chart repository, select the `local` cluster (the button with the bull's head), expand the `Apps` group and from the left menu click `Repositories`.

Then, click on the `Create` button on the top right of the page, and fill the form as follows:



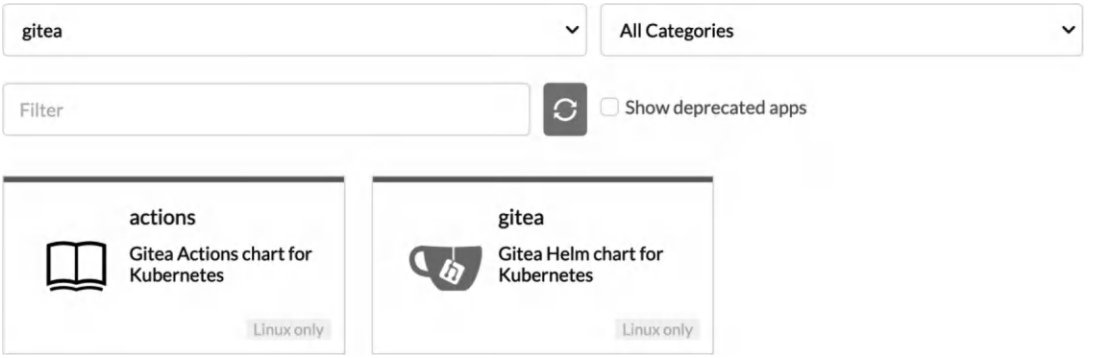
Once done, click on the `Create` button on the bottom right of the page to complete the process.

Deploy Gitea

We can now start to deploy Gitea: select the `local` cluster (the button with the bull's head), expand the `Apps` group, and click on `Charts` – this will display all the available charts.

Expand the first drop down box, which is used to filter the charts by repository, deselect `all` and select only the `gitea` repository.

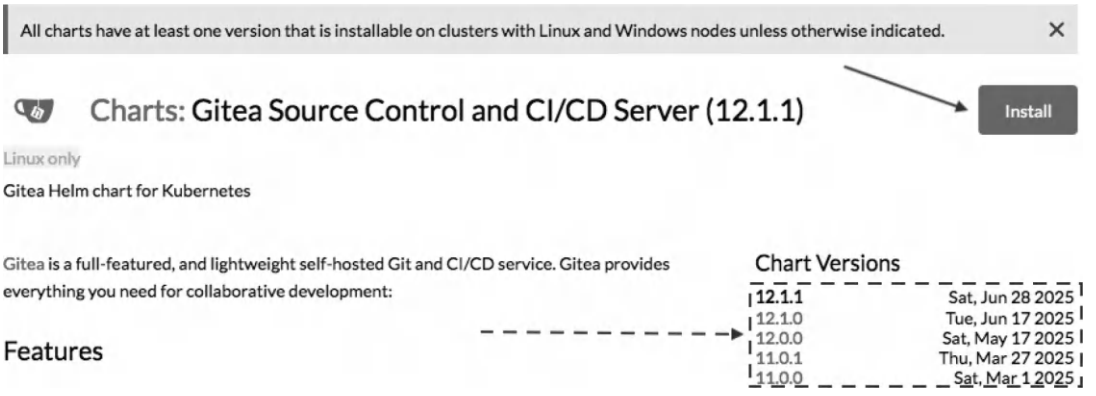
Once done, only the `actions` and `gitea` charts will be shown:



Click on the `gitea` chart to display its `Readme` page.

Note When deploying an App, or a Helm chart as well, always spend some time carefully reading the `Readme` page, since it provides the necessary documentation to guess the values that are necessary to be set to deploy the App so as to fit your scenarios as better as possible. Read it also when running an updated version of the chart, since it may have disruptive changes (they are supposed to be documented, if any).

As you can see from the below picture, the Readme page also provides the link to previous versions of this chart. In this example we are using `12.1.1`.



As a personal observation, it seems that the team developing the Gitea Helm chart has not yet established a clear manifest structure for the values file. Unfortunately, this sometimes leads to releases which break backward compatibility, creating challenges when upgrading the Helm chart.

For example, in Helm chart version `12.2.0`, they changed the naming convention for services, resulting in inconsistent service names. This is a compatibility-breaking change: deploying Actions Runners requires a stable service name, which was compromised in version `12.2.0`.

Note To avoid issues while completing the rest of the book, my suggestion is to stick with Helm chart version `12.1.1`: since this is a lab environment, using a fixed version – even if it becomes outdated – should not pose any problems. Once you become familiar with the setup, it will be easier to adapt to changes they will make in more current versions. To switch to this version, just select it from the Chart Versions list on the top right of the page – if it is not listed, click on show more to display all the available versions.

After selecting the chart version `12.1.1`, start the deploy process by clicking on the `Install` button on the top right.

When the wizard starts, the `gitea-t01` namespace should be automatically selected. If it isn't, select it, assign `gitea` as the name, and then click on the `Next` button.

Configure Helm Chart's Values

You must now configure the chart's values file to fit our scenario. To simplify this process, in the next paragraphs I am providing the reference snippets that highlight the parts of the YAML file that need to be changed.

Note To improve readability, each code snippet includes not only the values to add but also the full path to the relevant section in the YAML file. You must add these values to the appropriate section of the Helm chart's values.yaml file. If a section doesn't exist, create it – but be careful not to duplicate any existing sections.

Disable Valkey Cluster

Since we are in a lab, to keep things minimal, we disable the valkey cluster:

```
valkey-cluster:
  enabled: false
```

And valkey in Gitea as well:

```
valkey:
  enabled: false
```

Disable Dedicated PostgreSQL Instance

As anticipated, to minimize resource requirements, we are re-using the PostgreSQL instance from Pulp3.

The Gitea Helm chart provides a way for provisioning a PostgreSQL dedicated to Gitea: in our scenario we must keep them disabled.

To disable the deployment of a dedicated PostgreSQL-HA cluster, configure values as by the following snippet:

```
postgresql-ha:
  enabled: false
```

The Helm chart may contain also a snippet for a single instance PostgreSQL – it must be disabled as well as follows:

```
postgresql:
  enabled: false
```

Configure External PostgreSQL

You must now provide the settings for connecting to the database we created in the Pulp 3's PostgreSQL container.

These settings must be put beneath the `gitea.config` section, in the `database` dictionary.

You can use the following snippet as a reference:.

```
gitea:
  config:
    database:
      DB_TYPE: postgres
      HOST: 192.168.66.177
```

```
      NAME: gitea_t01
      PASSWD: grimoire
      SCHEMA: public
      USER: gitea_t01
```

Note The HOST variable contains the IP address of the Docker host. It is the same IP address you configured in the previous chapter for resolving the IP address of `rancher.cattle-system` in the `/etc/hosts` file (or `C:\Windows\System32\drivers\etc\hosts` if you are using Windows).

Configure Administrative Credentials

You must then configure Gitea's web UI administrative credentials and email address. Since you created a dedicated secret for them, you just need to specify that secret's name as `existingSecret` and configure the email address.

For example:

```
gitea:
  admin:
    existingSecret: gitea-admin-secret
    username: null
    password: null
    email: gitea_admin@carcano.corp
```

Server Specific Settings

There are also some server-specific configurations to be set:

```
gitea:
  config:
    server:
      LOCAL_ROOT_URL: http://gitea-http.gitea-t01.svc.cluster.local:3000
      DOMAIN: 192.168.66.177:30001
      SSH_DOMAIN: 192.168.66.177
      LFS_ALLOW_PURE_SSH: false
```

```
    LFS_START_SERVER: true
  lfs:
    PATH: /data/lfs
```

If you are using a namespace different from `gitea-t01`, please mind adjusting the `LOCAL_ROOT_URL` accordingly.

Note The `DOMAIN` and `SSH_DOMAIN` variable contains the IP address of the Docker host. It is the same IP you configured in the previous chapter for `rancher.cattle-system` in the `/etc/hosts` file (or `C:\Windows\System32\drivers\etc\hosts` if you are using Windows).

Configure Incoming Connections to Gitea

It is of course necessary to configure access to Gitea from outside Kubernetes. In this lab, to keep configurations at the bare minimum, we configure services of the kind `NodePort`. This Kubernetes service type enables exposing a Kubernetes service on specific ports of the running host.

Modify the values as by the following snippet, which exposes Gitea's `http` service on port `30001` and the `ssh` service on port `30002` on the Docker host:

```
service:
  http:
    nodePort: 30001
    port: 3000
    type: NodePort
  ssh:
    nodePort: 30002
    port: 22
    type: NodePort
```

Enable Gitea Actions Workflow

Finally, we must enable the support for Gitea Actions workflows. This feature enables external agents, called Gitea Actions Runners, to connect to the Gitea instance to run action-based workflows.

By enabling Actions Runners, we will be able to take advantage of Gitea's continuous integration (CI) features.

```
gitea:
  config:
    actions:
      ENABLED: true
```

Deploy Gitea

We are now ready to deploy Gitea. Just click on the `Install` button and wait for the deployment to complete – depending on your system and network speed it can take some time.

Note For your convenience, the `gitea-values.yml` file in this chapter's workspace directory contains the values that were used to deploy the Gitea instance while writing the book.

Log In to the Gitea Web UI

Upon completion, you can access the Gitea web UI with your web browser by pointing it to the IP address of your Docker host on port `30001`.

Note Mind that to keep everything simple, in this lab we are not using HTTPS.

For example:



click on the `Sign In` link on the top right of the page and enter the credentials you configured (in this example the user is `gitea-admin` with password `grimoire2025`).

Note Gitea is really an amazing piece of software: I strongly suggest you learn it by checking its official documentation at https://docs.gitea.com/next .

Deploy Gitea Actions Runners

As we said, Actions Runners are external agents which connect to the Gitea instance to run action-based workflows, unleashing Gitea's CI features.

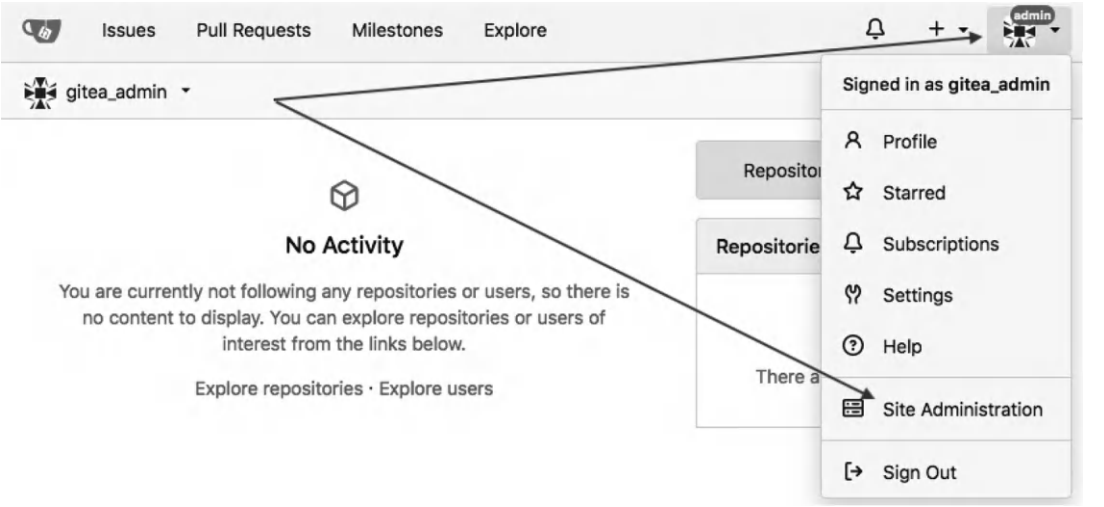
So far, we just enabled Actions Runners support in Gitea, but we still have not deployed any runner.

Generate an Actions Runner Registration Token

Actions Runners authenticate with the Gitea instance using a shared token. Before deploying any Actions Runner, it is necessary to get that token so to be able to provide it in its configuration.

Depending on where you generate the Runner's registration token, a runner can either be user-specific, organization level, repository level or global (instance) level, which affects the runner's scope and audience. In this lab we are setting up a Global runner.

Within Gitea's web UI, open the `Profile and Settings` menu by clicking on the icon on the top right of the page, and click on `Site Administration` .

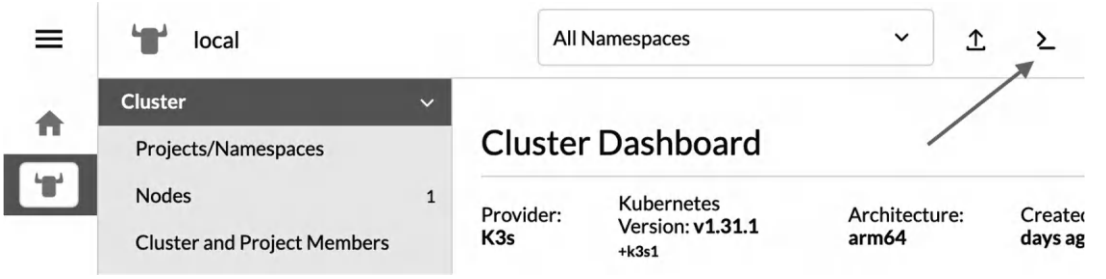


From the menu bar on the left, expand the `Actions` group and click on `Runners`.

This will display an empty table with the Gitea Actions Runners already registered. Click on the `Create new Runner` button and copy the displayed registration token.

Store the Token into a Kubernetes Secret

Leave the page open and, in another browser tab, from the Rancher's web UI, select the `local` cluster (the button with the bull's head) and click on the `Kubectl shell` button:



Using the just-opened text console, create the `token` key in the `gitea-actions` secret by running the following `kubectl` statement:

```
kubectl -n gitea-t01 create secret generic gitea-actions \
--from-literal=token=the_value
```

Of course, replace `the_value` with the registration token you just got from Gitea.

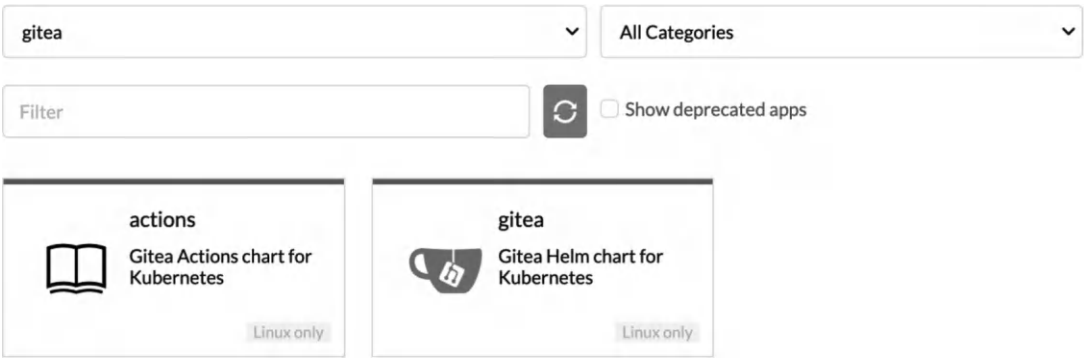
If you are using a namespace different from `gitea-t01`, please mind adjusting the namespace accordingly.

Deploy the Gitea Runner

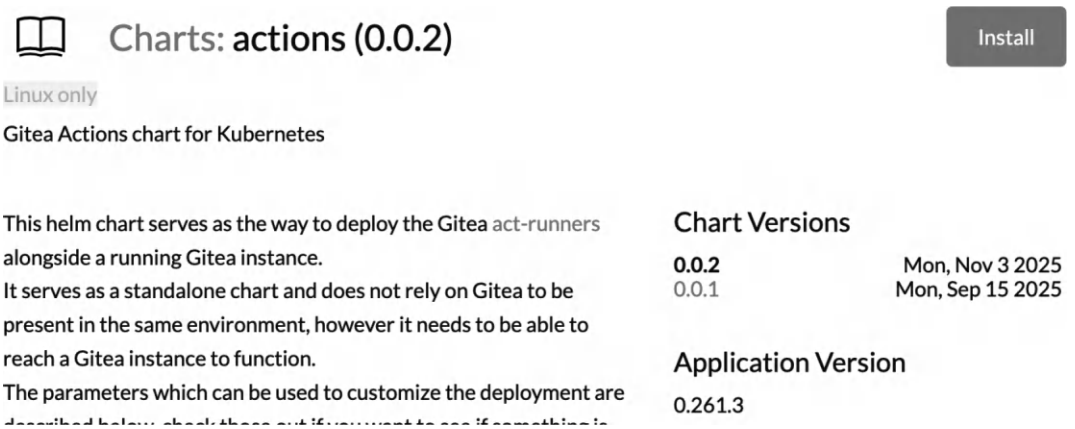
We can now start to deploy a Gitea Runner: select the `local` cluster (the button with the bull's head), expand the `Apps` group, and click on `Charts` – this will display all the available charts.

Expand the first drop down box, which is used to filter the charts by repository, deselect `all` and select only the `gitea` Repository.

Once done, only the `actions` and `gitea` charts will be shown:



Click on the `actions` chart to display its `Readme` page.



Spend some time reading it, then click on the `Install` button on the top right of the page.

Note I am describing the process for installing Gitea's Actions Runner workflow using the Helm chart version `0.0.2`: using different chart's versions may require adjustments to the described process.

Make sure the same namespace where Giteas was deployed (`gitea-t01` in this example) is selected and assign `actions` as the name. Then, click the `Next` button on the bottom right of the page: the values page will be displayed.

Just modify the values merging the contents of the following snippet:

```
enabled: true
existingSecret: gitea-actions
existingSecretKey: token
```

As value for the `giteaRootURL` attribute, specify `http://gitea-http.gitea-t01.svc.cluster.local:3000`.

If you are using a namespace different from `gitea-t01`, please mind adjusting the `giteaRootURL` accordingly.

We are now ready to deploy this Gitea Actions Runner. Just click on the `Install` button and wait for the deployment to complete – depending on your system and network speed it can take some time.

Note For your convenience, the `runner-values.yml` file in this chapter's workspace directory contains the values that were used to deploy the Gitea Actions Runner instance while writing the book.

Verify the Gitea Runner Registration

At deployment completion, within Gitea's web UI, you will see the runner listed in the table on Gitea's page where we got the registration token from:



A Quick Walkthrough with Gitea

We now have a running Gitea instance with a Gitea Actions Runner registered. Let's have a quick walkthrough to better understand how these components play together.

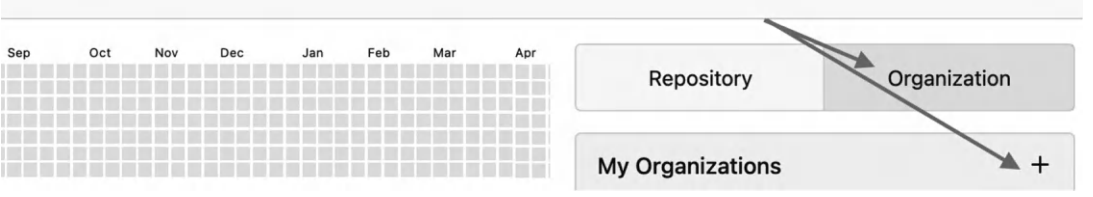
Create an Organization

Gitea organizations are security objects which enable the creation of organization-owned repositories and custom teams with specific access policies and grant access only to the specific organization's collaborators.

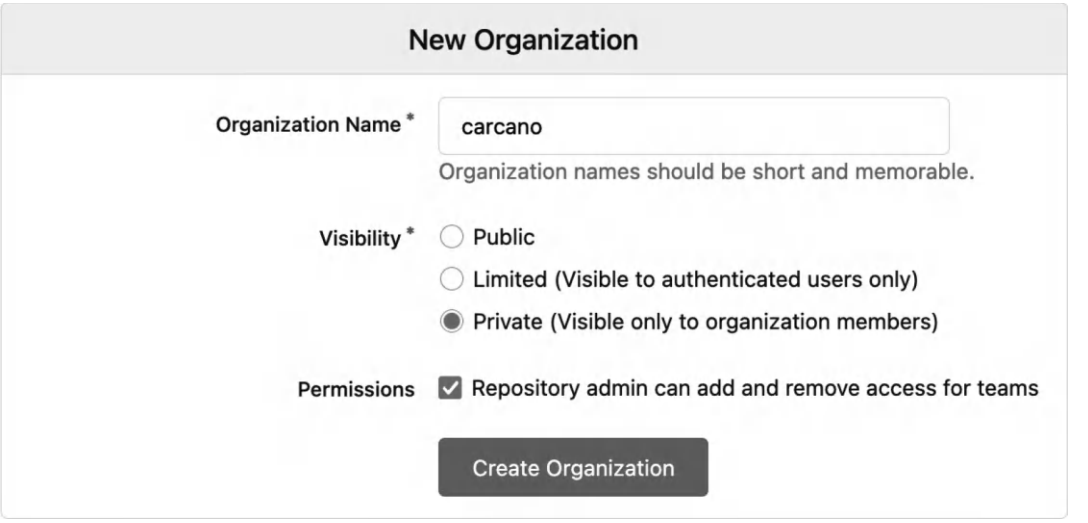
Let's start this lab by creating the `carcano` organization.

In Gitea's web UI, click on the cup of tea button in the top left corner of the page to display Gitea's splash page.

Then, on the top right of the splash page, select the `Organization` box, then click on the + symbol to open the `New Organization` form.



fill in the form as follows and click on the `Create Organization` button:

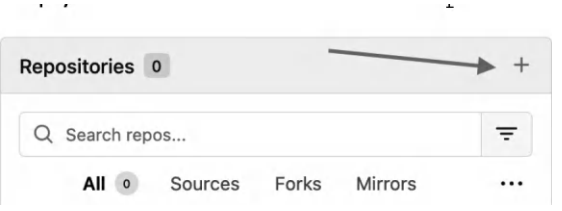


Create a Repository

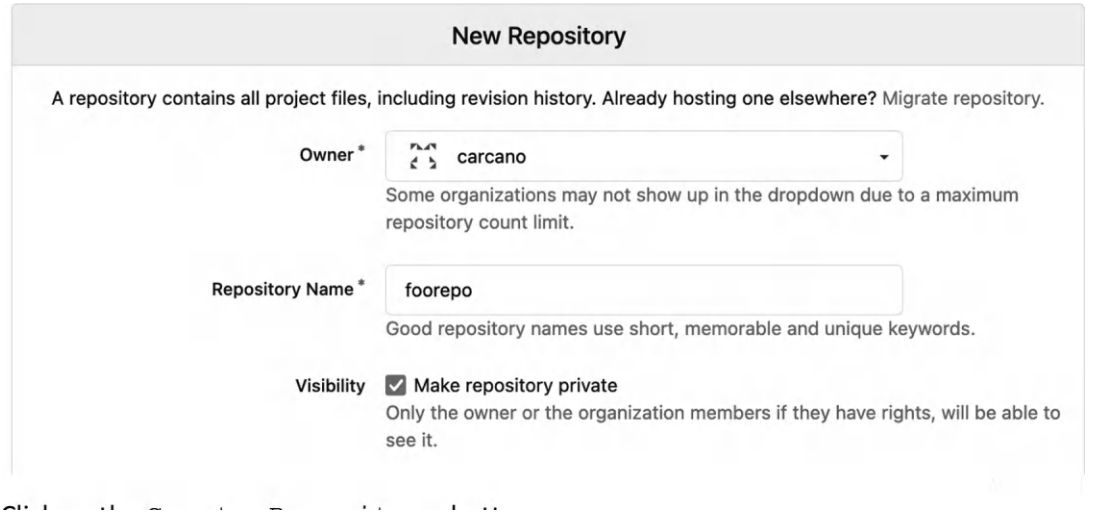
Gitea repositories can be owned either by a user or by an organization. In this last case, access to the repository is subject to the organization's policies.

Let's go on with the lab by creating a Git repository in the `carcano` Organization we just created.

Simply click on the + button in the `Repositories` box:



In this example we are creating the `foorepo` repository – complete the form as follows, remembering to tick the `Initialize Repository (Adds .gitignore, License, and Readme)` option:



Click on the `Create Repository` button.

Create an Actions Workflow

So far, we created the `foorepo` repository belonging to the `carcano` organization.

Let's complete the lab by creating a very basic Actions workflow just to see a first run with Gitea Actions.

The `onpush.yml` file in the chapter's workspace contains a very simple Gitea Actions workflow, triggered on push events, that just spin-up the Ubuntu container running a few simple statements on it. Its purpose is just displaying the use of some special variables and listing the repository's checked out files.

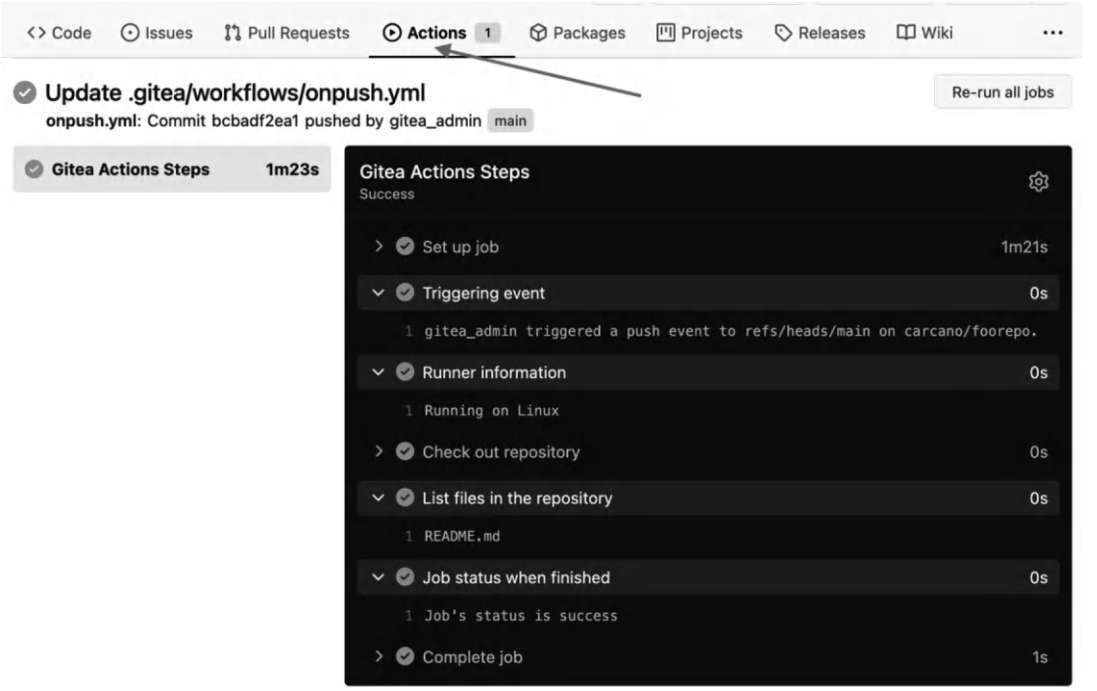
To configure the actions workflow, from Gitea's web UI, within the `carcano/foorepo` repository

- Click on the `Add File` button.
- Select the `New File` option.
- Create the `.gitea/workflows/onpush.yml` file using the contents of the `onpush.yml` file.

Once done, click on the `Actions` button: you will see the workflow we just defined already running.

Mind it takes some time since it needs to pull the Ubuntu container image before running the workflow using it.

At the end of the run, you can inspect each single workflow's step as shown in the below image.



From the above image you can see how the special variables are replaced, for example, `${{ gitea.actor }}` is replaced by `gitea_admin`, which is the user who triggered the push event.

Please mind that, if necessary, you can explore the contents of these special variables by dumping them to json using the `toJson` function. For example, `${{ toJson(gitea) }}`, `${{ toJson(runner) }}` and so on.

Note Gitea Actions workflows enable you to very easily set up a Continuous Integration within Gitea – I strongly suggest you dig more into this topic by reading their official documentation at https://docs.gitea.com/next/usage/actions/overview.

Gitea Helm Chart Caveats and Pitfalls

This chapter has shown you how to provision a hyperconverged solution for a small personal lab.

A real Helm-based deployment of Gitea is a quite challenging task, since there are several requirements and constraints that must be carefully analyzed and addressed.

This section illustrates some of these challenges and suggests possible solutions, providing some example snippets to be added to the Helm chart's value.

If you are updating an existing deployment, once changed the values as necessary, you must re-deploy the Helm chart.

Configure the Domain

Gitea requires setting the `DOMAIN` and `SSH_DOMAIN` variables with the FQDN or IP address of the Gitea endpoint. Those variables are used, for example, in the web UI when suggesting the URL for cloning repositories.

In this book, to keep requisites at bare minimum, avoiding setting up an external DNS entry for the Gitea service, we set it with Gitea's endpoint IP address (which in this case matches the Docker host's IP address). Since we exposed the service on a non-default port, we also included the port number.

Although this setup can work for a personal lab, it's unsuitable for a production system.

If you are configuring a regular Gitea instance, you must set the `DOMAIN` and `SSH_DOMAIN` variables with Gitea's endpoint FQDN, having it first registered in the corporate's DNS as well.

For example, if the Gitea instance's FQDN is `gitea-t01.carcano.ch`:

```
gitea:
  config:
    server:
      DOMAIN: gitea-t01.carcano.ch
      SSH_DOMAIN: gitea-t01.carcano.ch
```

Configure Webhooks

Webhooks enable Gitea to integrate with other software by creating broker services that are called by the webhooks on specific events. The call involves sending a JSON document with all the necessary details to describe the event which triggered the call.

For security reasons, webhooks endpoints must be whitelisted by configuring the `ALLOWED_HOST_LIST` variable.

For example:

```
gitea:
  config:
    webhooks:
      ALLOWED_HOST_LIST: *.carcano.corp
```

The value is a comma-separated list of FQDNs (the use of the * wildcard is supported), IP addresses, or keywords:

- `loopback`: 127.0.0.0/8 for IPv4, ::1/128 for IPv6, localhost included.
- `private`: RFC 1918 (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16) and RFC 4193 (FC00::/7)
- `external`: any valid non-private unicast IP

Configure Migration Sources

In real life, it is quite rare to set up Gitea and just start with brand new Git repositories: most of the time you are instead installing Gitea to migrate existing Git repositories from another SCM.

In such a scenario, you must add the list of the SCM servers containing the repositories to be migrated to the `ALLOWED_DOMAINS` variable. You can set specific FQDNs or use the * wildcard character to allow wider matches.

For example, to authorize migrations from any server of the `carcano.corp` domain:

```
gitea:
  config:
    migrations:
      ALLOWED_DOMAINS: '*.carcano.corp'
```

Gitea Version

After some time, you will need to update the Gitea instance to a more current version. Deploying a new Gitea version does not require updating the Helm charts.

The Helm charts are just the deployment automation: they must be updated too as necessary, but their versioning is unrelated to the Gitea one.

The Gitea version to deploy is set by the container's image tag.

For example:

```
image:
  tag: 1.24.2
```

Load Balancers

In our lab, to keep configuration and requirements minimal, we haven't installed any Load Balancer application, such as `MetalLB`. As a result, we couldn't configure a service of type `LoadBalancer`.

In a regular Gitea deployment, unless you want to set up an external load balancer (with all the additional maintenance effort it requires), it is instead necessary to set up a service of type `LoadBalancer`. In addition to that, since the HTTPS and SSH endpoint must be reachable at the same FQDN, they must share the same `LoadBalancer`.

This sharing requirement prevents using an Ingress rule on the default Ingress controller: the default Ingress controller runs indeed in a namespace different from the one used by Gitea, and Kubernetes forbids sharing a `LoadBalancer` service between Pods belonging to different deployments or namespaces.

In such a scenario, a dedicated service of the kind `LoadBalancer` is the only way out.

SSH Endpoint

You can use the following snippet as a reference for configuring the SSH endpoint on the Load Balancer:

```
service:
  ssh:
    annotations:
```

```
      metallb.universe.tf/allow-shared-ip: gitea-t01
    loadBalancerIP: 10.11.12.13
    port: 22
    type: LoadBalancer
```

The `metallb.universe.tf/allow-shared-ip` is used to provide a label for sharing this Load Balancer's IP address with others having the same label.

Just remember changing the IP address with the one assigned to the Load Balancer and to set as `metallb.universe.tf/allow-shared-ip` a proper identifier suitable to your use case in place of `gitea-t01`.

HTTPS Endpoint

You can use the following snippet as a reference for configuring the HTTP endpoint on the Load Balancer:

```
service:
  http:
    annotations:
      metallb.universe.tf/allow-shared-ip: gitea-t01
    loadBalancerIP: 10.11.12.13
    port: 443
    type: LoadBalancer
```

The IP address, as well as the value of `metallb.universe.tf/allow-shared-ip`, must match the ones you defined for the SSH endpoint.

Note As you can see port is set to `443`, which is the `https` default port: in this lab, to keep things simple we didn't, but in a real-life scenario, the HTTP endpoint must always be protected using TLS.

TLS Certificate

As we said, you must always protect communications by enabling TLS.

Since in this setup the Gitea services are directly exposed by the Load Balancer service, clients directly communicate with the Gitea service.

This means that this TLS certificate must

- Be valid for the `DOMAIN` and `SSH_DOMAIN` you configured
- Be signed by a CA trusted by your clients

In addition to that, the certificate's file must also include the intermediate CA's certificates.

Once you have the X.509 certificate and private key matching the above requirements, you must create a secret in the same namespace of your Gitea instance and upload the TLS certificate and key to it.

Once done, you must mount the certificate and private key in Gitea's container.

For example, if the secret is `carcano.ch-cert`

```
extraVolumeMounts:
  - name: tls-server
    subPath: tls.crt
    mountPath: /etc/gitea/tls.crt
    readOnly: true
  - name: tls-server
    subPath: tls.key
    mountPath: /etc/gitea/tls.key
    readOnly: true
extraVolumes:
  - name: tls-server
    secret:
      items:
        - key: tls.key
          path: tls.key
        - key: tls.crt
          path: tls.crt
      optional: false
      secretName: carcano.ch-cert
```

In addition to mounting the files in the container's filesystem, you must also configure Gitea to use this TLS certificate.

Just add the path to these files as environment variables, as shown by the following snippet:

```
gitea:
  config:
    server:
      CERT_FILE: /etc/gitea/tls.crt
      KEY_FILE: /etc/gitea/tls.key
      LOCAL_ROOT_URL: https://gitea-t01.carcano.ch/
      PROTOCOL: https
      ROOT_URL: https://gitea-t01.carcano.ch/
```

Actions Runners with TLS-Enabled Gitea Endpoint

Working with TLS requires some fixes on Actions Runner configuration depending on whether you are using a global CA or a private CA.

If the TLS certificate is signed by a global CA, you must configure the Gitea Actions Runners to connect to the gitea endpoint FQDN by setting the `GITEA_INSTANCE_URL` variable.

For example:

```
statefulset:
  actRunner:
    config: |
      runner:
        envs:
          GITEA_INSTANCE_URL: "https://gitea-t01.carcano.ch"
```

If the TLS certificate is signed by a private CA, you must configure Gitea Actions Runners to trust it.

Since writing this book it seems there is no such setting in the runner configuration file; the only other way out is to configure the runners to skip the TLS check by setting the `insecure` key.

For example:

```
statefulset:
  actRunner:
    config: |
      runner:
        insecure: true
```

Note As usual, do not disable TLS verification unless you have no other way out, and only after having understood all the security implications that exception causes to that specific use case.

Actions Runner Tweaks

You may want to fine-grain tune your Gitea Actions Runners by settings:

- `capacity`: the number of parallel jobs the Actions Runner bears
- `timeout`: the job completion timeout
- `fetch_timeout`: timeout for fetching an Action from Gitea
- `fetch_interval`: interval between pools to the Gitea endpoint

For example:

```
statefulset:
  actRunner:
    config: |
      runner:
        capacity: 1
        timeout: 3h
        fetch_timeout: 5s
        fetch_interval: 2s
```

Configuring Logging

A proper logging configuration is essential in every setup, since it dramatically helps you quickly figure out what's happening. This is particularly useful when problems arise, and you must very quickly figure out their root cause.

Gitea

The following snippets provide a convenient logging configuration you can easily tweak for your specific use cases.

First, configure the router to log on the `console` as shown by the following snippet:

```
gitea:
  config:
    log.logger.router:
      MODE: console
```

Then, configure the main logger, as shown by the following snippet:

```
gitea:
  config:
    log:
      LEVEL: Info
      MODE: console,traceconsole
      MACARON: console
      REDIRECT_MACARON_LOG: true
```

Note Valid LEVEL values are Trace, Debug, Info, Warn, Error, Critical, Fatal and None.

The above snippet configures two logging consoles: `console` and `traceconsole`: `console` is used for the standard logging, whereas `traceconsole` is used to log only specific events, configuring severity and matching criteria as necessary.

The following snippet provides a stub for configuring the `traceconsole` with never-matching criteria:

```
gitea:
  config:
    log.traceconsole:
      MODE: console
      LEVEL: trace
      EXPRESSION: NothingForNow
```

Note When you need to trace anything, just set `EXPRESSION` to what you want to match and run a re-deploy of the `gitea` deployment. For example, if you are troubleshooting LDAP problems, configure `EXPRESSION: ldap`.

Actions Runners

If you are using Actions Runners, problems may arise with them as well. To figure out what they are doing, you must configure logging for them as well.

You can use this snippet as a reference:

```
statefulset:
  actRunner:
    config: |
      log:
        level: info
```

When necessary, just raise the log level and redeploy the runner.

LDAP Authentication

Real Gitea deployments are multi-user: for this reason, Gitea supports external authentication sources, such as OAuth or LDAP.

The following snippet provides an example LDAP/Active Directory configuration you can use as a template as necessary:

```
gitea:
  ldap:
    - name: carcano.corp
```

```
host: ldap.carcano.corp
port: 636
bindDn: CN=gitea-t01,OU=Accounts,DC=carcano,DC=corp
bindPassword: theSecurePassword
emailAttribute: mail
publicSSHKeyAttribute: publicSSHKey
securityProtocol: ldaps
userFilter: (&(sAMAccountName=%s)(memberOf=CN=gitea-t01,OU=Groups,
DC=carcano,DC=corp))
adminFilter: (memberOf=CN=gitea-admins,OU=Groups,DC=carcano,DC=corp)
userSearchBase: OU=Accounts,DC=carcano,DC=corp
usernameAttribute: sAMAccountName
```

The available keys are the ones used by the gitea admin command-line utility described at https://docs.gitea.com/administration/command-line#admin.

Sadly, since that utility does not allow you to configure every available LDAP option, once the deployment is completed, you must log in to the web UI and complete the configuration by manually setting the following:

- **First name attribute**: `cn`
- **Surname attribute**: `sn`
- **Avatar attribute**: `thumbnailPhoto`

You must also tick the `Enable LDAP Groups` and add

- **Group search Base DN**: OU=Groups,DC=carcano,DC=corp
- **Group attribute containing the list of users**: member
- **User attribute listed in group**: dn

Gitea also supports the automatic creation of teams into organizations, mapping LDAP groups to them. If you want to configure this synchronization, you must also configure `Map LDAP groups to Organization teams`.

This mapping is configured using a JSON document with the format illustrated by the following snippet:

```
{
  "CN=gitea-revisor,OU=Groups,DC=carcano,DC=corp": {
```

```
        "FooOrg": [
            "Revisor"
        ],
        "BarOrg": [
            "Revisor"
        ]
    },
    "CN=gitea-maintainer,OU=Groups,DC=carcano,DC=corp": {
        "FooOrg": [
            "Maintainer"
        ],
        "BarOrg": [
            "Maintainer"
        ]
    }
}
```

In this example, the `gitea-revisor` LDAP group is mapped to the `Revisor` team in the `FooOrg` and `BarOrg` organizations, and the `gitea-maintainer` LDAP group is mapped to the `Maintainer` team in the `FooOrg` and `BarOrg` organizations.

Enable Monitoring

As with every Kubernetes workload, the best practice is to enable Prometheus metrics to be able to display them in a Grafana Dashboard.

The following snippet shows how to enable exporting Prometheus metrics:

```
  metrics:
    enabled: true
    serviceMonitor:
      enabled: true
      scheme: https
      tlsConfig:
        insecureSkipVerify: true
```

Note Since Prometheus connects to the ServiceMonitor using the IP address, even if the certificate is valid, we must disable TLS verification because the certificate does not contain that IP address.

Besides enabling Service Monitor, it is also necessary to enable metrics support in the Gitea configurations, as shown by the following snippet:

```
gitea:
  config:
    metrics:
      ENABLED: true
      ENABLED_ISSUE_BY_LABEL: true
      ENABLED_ISSUE_BY_REPOSITORY: true
```

If you want to enable also performance metrics

```
gitea:
  config:
    server:
      ENABLE_PPROF: true
```

Configure Resource Requests and Limits

In all crowded shared environments, hardware resources are precious and must be carefully managed. This is typically employed by capping them at the project level and at the namespace level.

It is also important to specify the minimum amount of resources necessary to start applications, just to avoid having them scheduled on hosts with not enough resources: if this happens, the likely outcome is a crash-looping application consuming the few leftover resources at each restart, causing resource contentions to the other applications leading to instability.

The following snippet shows how to configure the minimum number of resources required for starting Gitea and the resource limits:

```
resources:
  limits:
```

```
    cpu: 2000m
    memory: 4096Mi
  requests:
    cpu: 100m
    memory: 512Mi
```

Note Mind that, conversely from the cpu limit, which throttles the cpu usage, when the memory limit is reached, the container gets OOM killed.

Scaling

If you fancy, it is possible to horizontally scale the Gitea deployment increasing the number of replicas.

This, besides altering the number of replicas, requires also

- Using a shared storage
- Using a Redis instance for storing caches, queues and sessions
- Configuring session affinity on the `gitea-http` service, setting it to Client IP

Number of Replicas

The number of replicas of the Gitea Pod is governed by the `replicaCount` value.

For example:

```
replicaCount: 2
```

Shared Storage

Configuring the shared storage requires pre-provisioning a PV on a NAS or SAN or using a CSI operator for provisioning it on the fly.

As for the Helm values to set, you can use the following snippet as a basic reference:

```
persistence:
  accessModes:
ReadWriteMany
  annotations:
    helm.sh/resource-policy: keep
```

Caches, Queues, and Sessions

As we said, we must also configure caches, queues, and session to use Redis. Redis is an in-memory key-value store that has been specifically designed for use cases like this.

Note Although the Gitea helm chart can deploy dedicated PostgreSQL and Redis instances, my suggestion is to deploy Redis (or KeyDB) using their own Helm chart. As for PostgreSQL, you can use an external existing PostgreSQL cluster or deploy it on Kubernetes as well. In this case, you may want to try EDB's Helm chart or CloudNativePG.

A Redis instance is referenced by providing a URL such as:

```
gitea@keydb-headless.keydb-t1.svc.cluster.local:6379/0
```

This example refers to an existing keydb instance that has been deployed in the `keydb-t1` namespace.

You can use the following snippet as a reference for configuring cache:

```
gitea:
  config:
    cache:
      ADAPTER: redis
      ENABLED: true
      HOST: >-
        redis://<REDIS_URL>
```

This other snippet instead shows how to configure queues:

```
gitea:
```

```
  config:
    queue:
      CONN_STR: >-
        redis://<REDIS_URL>
```

This last snippet is instead for configuring sessions:

```
gitea:
  config:
    session:
      PROVIDER: redis
      PROVIDER_CONFIG: >-
        redis://<REDIS_URL>
```

Summary

This chapter provided hands-on experience with Rancher, focusing on deploying a complex solution like Gitea and Gitea Actions Runners. You learned to navigate the entire deployment process, starting from meeting the necessary requirements. In addition to the lab exercises, the chapter offered valuable insights into the challenges and pitfalls you may encounter when deploying Gitea in real-world scenarios.

The skills acquired in this chapter will enable you to deploy applications on Rancher using Helm charts, a highly sought-after competency for expert DevOps and DevSecOps professionals. Deploying applications on Rancher is crucial, as it simplifies the management of containerized applications, allowing teams to orchestrate and scale deployments efficiently. This capability enhances operational efficiency and fosters collaboration between development and operations teams, leading to faster software delivery.

Understanding how to deploy Gitea and Gitea Actions Runners is equally significant, as Gitea serves as a lightweight, self-hosted Git service that supports version control and collaboration. Gitea Actions Runners facilitate continuous integration and deployment, enabling teams to automate workflows and maintain high-quality code. Together, these tools empower teams to build secure and reliable applications rapidly, reinforcing the core principles of DevOps and DevSecOps.

The chapter also introduced Gitea Actions workflows, laying the groundwork for further exploration of this critical topic, as workflows are essential for constructing continuous integration pipelines.

Overall, the knowledge gained from this chapter equips you with the necessary skills to thrive in a DevOps and DevSecOps environment.

CHAPTER 33

CI/CD and Continuous Delivery

In traditional development, integrating code from multiple developers can lead to conflicts and bugs that inevitably delay release cycles.

This is a huge business problem, since long release cycles can hinder a team's ability to respond to user feedback and market changes.

Another problem of traditional development is manual deployment processes: these are too error-prone and inconsistent. From the business perspective, this means not only that they are not cost-effective, but also that they are risky, generating problems spanning from not earning money during service outages to reputation loss and raising of damages requests from customers.

This chapter illustrates how to mitigate these problems using a CI/CD toolchain: after introducing the concept, it discusses the various components of a CI/CD toolchain, such as the SCM, repository managers, the build automation facilities, and the test automation facilities. Moreover, the chapter describes the CI and CD process highlighting the key responsibilities of each involved team. The chapter ends explaining the continuous deployment process.

Having a good understanding of the CI/CD toolchain and CI/CD processes is a critical skill for DevOps and DevSecOps professionals, since it enables them to streamline workflows and enhance collaboration among teams and ensures secure, efficient, and automated software delivery, ultimately contributing to the success of their organizations.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_33

This chapter provides a brief, blended summary of concepts drawn from established DevOps practices and methodologies, including continuous integration/continuous delivery (CI/CD) frameworks. It is intended solely as an introductory overview to explain the existence and basic application of these practices in a DevOps and DevSecOps context. These concepts need to be introduced to help the user make sense of the labs in the next chapters.

This is not a comprehensive or official guide, and it does not replace the original sources. For a deeper understanding, I strongly recommend consulting the primary materials directly:

- **For introductory DevOps and CI/CD concepts**: *DevOps for Dummies* by Emily Freeman and Joel Bancroft-Connors
- **For CI/CD and toolchain concepts**: *The DevOps Handbook* by Gene Kim, Jez Humble, Patrick Debois, and John Willis
- **For continuous delivery practices**: *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* by Jez Humble and David Farley

Readers are encouraged to explore these works for accurate, detailed information.

CI/CD

Continuous integration, continuous deployment, and continuous delivery are a set of practices and tools aimed at addressing all these problems, improving the software development process by automating the integration, testing, and deployment of code changes.

The automated integration of code changes, running tests, and deploying applications reduces both manual effort and human error. In addition to that, by streamlining the development process, CI/CD enables teams to deliver new features and updates to users more quickly and efficiently, resulting in rapid feedback to developers about the state of their code, allowing them to address issues early in the development cycle.

Moreover, automated testing ensures that code changes do not introduce new bugs, maintaining the quality of the software.

The CI/CD Toolchain

The CI/CD process employs a range of tools, often delivered as services, to facilitate collaboration between developers and quality assurance (QA) professionals.

Like cloud services, CI/CD adheres to a shared responsibility model, where the responsibilities for managing the toolchain are distributed among various stakeholders.

DevOps professionals are primarily responsible for the deployment, configuration, maintenance, and support of the CI/CD toolchain. Their involvement may vary based on organizational structure and project requirements; they may work closely with development and QA professionals or be part of a dedicated team focused solely on CI/CD operations.

Note The reliability and availability of CI/CD components are supported by operations teams, including infrastructure and monitoring teams. These teams ensure that services remain operational and compliant with Organizational Level Agreements (OLAs), which define the expected service levels and performance metrics.

This section illustrates the basic components of this toolchain:

- Source Code Management (SCM)
- Repository manager
- Build automation facilities
- Automated testing frameworks

Source Code Management (SCM)

This is the first component of a software toolchain: Source Code Management (SCM) refers to tools that facilitate the management of source code changes and version control. These tools extend the capabilities of Version Control Systems (VCSs) like Git by providing a web-based user interface (UI) and advanced collaborative features such as pull requests, approval gates, issue tracking, and more.

Modern SCM platforms, such as GitLab and Gitea, not only support version control but also integrate continuous integration (CI) features, allowing for automated testing, building, and deployment processes within the software development lifecycle.

Repository Managers

A repository manager is software that facilitates the management of one or more repositories, enabling the publishing and retrieval of artifacts using various protocols such as HTTP/HTTPS, FTP, SFTP, and others.

Many repository managers support a variety of technology-specific repositories, including YUM, APT, PyPI, and Vagrant, as well as container images, thereby functioning as container image registries.

Popular repository managers include Sonatype Nexus, JFrog Artifactory, and Pulp3. Additionally, some Source Code Management (SCM) tools, such as GitLab and Gitea, offer support for creating both generic and technology-specific repositories.

Repositories

A repository serves as the distribution point for artifacts (software packages, compressed archives, container images, and so on).

They can be set up to provide

- Locally stored artifacts
- Mirrored online repositories, pulling the artifacts on the fly and caching them for next requests or periodically synchronizing artifacts (all or the most recent versions only) from the linked online repository

Repositories can be categorized into two main types:

- **General-purpose repositories**: These repositories contain a mixed variety of package formats and are not limited to a specific technology.
- **Technology-specific repositories**: These repositories are dedicated to a particular package format. For example, a `yum` or `dnf` repository exclusively contains RPM packages.

The primary advantage of technology-specific repositories over general-purpose ones is their ability to provide format-specific metadata. This metadata enables package client-side management utilities to perform online queries not only about package availability but also about their contents and dependencies. For instance, both `yum` and `dnf` support detailed queries, such as identifying which RPM package provides a specific file or installing a specific package group, which is automatically expanded into the relevant list of RPM packages.

To achieve this, technology-specific repositories typically provide specific tools for scanning the software within the repository and generating the necessary metadata.

Note The automatic dependency-solving feature provided by client-side package management utilities of technology-specific repositories is a double-edged sword that can cause significant issues if the distributed packages of different versions have incompatible requirements. One way to mitigate this problem is by grouping the packages in repositories that match the same Linux distribution family, hardware architecture, and the application's major or minor version, depending on the level at which incompatible requirements are expected to arise.

YUM and DNF Repositories

RPM packages are typically published using `yum` or `dnf` technology-specific repositories.

Whereas the `yum` command-line utility can use both kinds of repositories, it does not support `dnf`-specific features such as module streams, which instead require the `dnf` command-line utility.

Note Module streams are an extension to the `yum` feature set which allows users to manage different versions of software packages. This feature enables users to select specific streams of packages based on their need.

The metadata about files provided by specific packages and package's dependencies are generated (or re-generated) using the `createrepo` command-line utility while using `yum`, whereas `dnf` repositories utilize the `createrepo_c` utility.

Metadata which cannot be automatically inferred by scanning the RPM packages, such as package groups and modules, require instead manual configuration through specific definition files:

- **Package group metadata**: This metadata is generated by reading the contents of any file with a `.xml` extension located in the directory specified using the `--groupfile` option.
- **Module metadata**: This metadata is generated by reading the contents of any file with a `.md` extension located in the directory specified using the `--modulemd` option.

APT Repositories

An APT repository is a collection of software packages for Debian-based systems, providing binary and source packages along with metadata for installation and dependency management.

They can be created by running utilities such as `reprepro`.

Container Image Repositories

We already discussed them, so we won't repeat what we already said – we listed them here only to remark that they are technology-specific repositories that can be hosted not only by container image registries but also by repository managers, if they implement the container image registry interface.

PyPI Repositories

As we already mentioned, a PyPI repository is a centralized collection of Python packages that includes metadata such as package name, version, author, and dependencies.

Note There are other examples of technology-specific repositories, such as Apache Maven for Java projects, npm for JavaScript packages, Ansible Galaxy for Ansible roles and collections, Vagrant image repositories, and so on.

Build Automation Facilities

Build automation is typically supported by a set of software tools and facilities that operate on purpose-specific machines (often referred to as build servers or build agents) equipped with suitable hardware to ensure that builds are completed efficiently and within a reasonable timeframe that aligns with business needs.

If it is required to support multiple architectures, these often make use of cross-compilers or run the build emulating a platform/architecture environment using QEMU with Docker as necessary, falling over to hardware devices (such as embedded devices) only as the last resort.

To decouple the build from the testing process, they run in distinct stages:

- The build process compiles the sources as necessary, creates software packages, and publishes them into the appropriate software repository.
- The automated testing facilities retrieve the just-created snapshot packages, install them, and run the tests.

When dealing with software for embedded devices, the deliverable packages often include not just software packages but also OS images or container images that encapsulate the application along with all its dependencies.

In scenarios where the deliverable is an OS bootable image, the image is typically built into a target directory tree using tools such as `debootstrap` or `yum --instroot`, with the target architecture emulated using QEMU as needed. The installation of previously built packages and the associated configuration process within the target directory tree can be automated using configuration management tools like Ansible. Once the process is complete, the OS bootable image is generated and stored as a generic artifact, either in snapshot or release candidate repositories, depending on the use case.

If the deliverable is a container image, then the Docker build process is usually employed, with QEMU used for emulating the target architecture when necessary. After the build process is complete, the container image is pushed to the appropriate container image registry, either in snapshot or release repositories.

Automated Testing Facilities

Automated testing facilities are provided by a set of software tools that automatically install the specific application's packages on a runtime platform-architecture matching the ones the package is supposed to run on or pull the container images containing the application to be tested.

Once done, the testing facilities take care to run a multistage pipeline with a stage for vulnerability tests, a stage for unit tests, a stage for component tests, and a stage for integration tests. The failure of a stage will cause the abortion of the pipeline.

To speed up things while testing, it is sometimes possible to implement fast-feedback loops which enable skipping the component testing and, after passing unit tests, going straight to the integration tests.

Note The above description represents the simplest possible continuous integration (CI) pipeline: more thorough CI pipelines also include a stage for quality checks (linting and static code analysis) and code coverage reports and parallelize the stages whenever possible. It is also quite common to implement a continuous delivery (CD) pipeline, triggered after the completion of the CI pipeline, which delivers the tested application to the staging environment. In addition to that, they often implement feedback reporting, such as mail messages to those who started the process.

When running unit tests, it is crucial that these tests are self-contained and do not depend on external services. Therefore, mocks or stubs should be used to simulate the behavior of external services, ensuring that no real calls occur.

Real calls are instead implemented by component tests: Gatehill's `Imposter` can be a great tool for component testing, where interactions with simulated services are necessary.

When dealing with embedded devices, the deployment process of the OS images or container images is often addressed using specific tools for managing over-the-air (OTA) software updates such as `Mender`.

Continuous Integration (CI)

Continuous integration (CI) is the practice of frequently integrating code changes into a shared repository, which involves continuously iterating through the following processes:

- Development
- Continuous building
- Continuous testing

Development

This is the process of writing and committing code changes to the version control system.

The professionals involved are developers, quality assurance, and DevOps.

Developers

Developers are responsible for developing applications by adding features while ensuring that unit tests are provided to cover all feature code. They also implement component tests: these tests, used to validate the interfaces, often rely on mock services, allowing for early identification of issues before consuming a real environment for running integration tests. Additionally, developers fix bugs as necessary and write integration tests, which validate the end-to-end interactions between different components of the application.

Developers also create and maintain documentation related to their code, which includes valuable information for QA professionals when writing additional integration tests and for monitoring teams when setting up probes, alarms, and sanity checks.

Furthermore, developers configure build automation tools (such as Makefiles when using GNU Make) to provide targets not only for building the code but also for running unit tests and component tests.

In the workflow, developers raise pull requests (PRs) to merge changes from individual feature branches into long-lived branches. This process may include cherry-picking specific commits to ensure that only relevant changes are included. Once a pull request has been reviewed and approved by team leads or peers (often including input from QA professionals for testing validation), changes are merged into the target branch.

Quality Assurance

The primary responsibility of the QA professionals centers on ensuring code quality throughout the development lifecycle, particularly during pull request (PR) reviews.

Their core function is being a critical quality gatekeeper for merges: they verify that all changes submitted in a PR are thoroughly tested and meet the defined quality standards for the target branch. This includes evaluating whether the code passes required test suites, adheres to performance benchmarks, and complies with organizational coding practices before providing approval or recommendations.

Depending on the organization's practice, QA professionals often extend their role into test authorship, writing supplementary integration tests to cover edge cases, user workflows, or scenarios not fully addressed by developer-written tests. These tests evolve alongside the application, requiring ongoing refinement to adapt to new features, architectural changes, and shifting requirements.

DevOps

The primary responsibilities of the DevOps professionals include setting up and maintaining build automation facilities, which involves configuring integrations to enhance the existing build automation tools created and maintained by developers.

In addition to building automation, DevOps professionals establish test automation facilities by integrating the integration tests developed by developers and quality assurance (QA) professionals into continuous integration and deployment (CI/CD) pipelines.

Furthermore, DevOps professionals are responsible for implementing monitoring and reporting tools that track the performance and success of both builds and tests. This monitoring provides valuable feedback to Developers and QA professionals, enabling them to identify issues early in the development process and ensure that the software meets quality standards before deployment.

Continuous Building

This process involves automatically building the application to ensure that the code compiles and integrates correctly.

The continuous building pipeline typically includes running unit tests and component tests to verify that the application behaves as expected after each build.

The advantage of running component tests, postponing integration tests at the next stage only if they pass is that component tests deliver quicker and more segregated insights, sparing to use real services only when the code is good enough to deserve running real end-to-end tests.

Table 33-1 depicts the specific tasks of each professional in the continuous building process.

Table 33-1. *Professionals and tasks in the continuous building process*

Professionals	Tasks
Developers	Run automated builds of the code in their feature branches, fixing it as necessary until there are no longer failing unit tests or component tests out of the ones tracked as exceptions. When the code of their branch is ready to be merged, they raise a pull request from it.
DevOps	Maintain the automated testing facilities and infrastructure that support the deployment of integration services for running component tests. Additionally, DevOps monitors performance metrics to ensure they align with expectations outlined in Service Level Agreements (SLAs) or Operational Level Agreements (OLAs). In the event of anomalies, DevOps addresses platform-related issues and collaborates with the operations team for vertical scaling or provisioning resources for horizontal scaling as needed.

Continuous Testing

This process entails executing automated checks to confirm the performance and reliability of the code.

The Continuous Testing pipeline generally incorporates executing unit tests and integration tests. Conversely from component tests, which aim to verify standalone parts, frequently including certain dependencies like simulated services, integration tests perform real end-to-end tests, using real services.

That said, using integration tests after and as a complement to component tests provides a thorough validation, which is particularly valuable in intricate setups where spotting problems early is essential.

Table 33-2 depicts the specific tasks of each professional in the Continuous Testing process.

Table 33-2. *Professionals and tasks in the continuous testing process*

Professionals	Tasks
Developers	Execute unit tests and integration tests to obtain rapid insights into integration issues and code reversions in their work prior to submitting merge proposals.
Quality Assurance	Execute unit tests and integration tests on branches with pending merge proposals. This encompasses monitoring results and resolving any failures. QA teams work hand in hand with developers to provide feedback on test outcomes, debate problems, and verify that the checking process fits the broader development cycle. Moreover, QA specialists confirm that software standards satisfy predefined quality gates, encompassing elements like code style checks, static code analysis, and test coverage metrics. They strive to keep technical debt low, promoting a codebase that is easy to maintain and high in quality.
DevOps	Oversee the automated checking tools and add-ons that enable the seamless setup of integration environments for test execution. Furthermore, DevOps experts track system performance indicators to match the benchmarks in the Organizational Level Agreement (OLA). When anomalies arise, DevOps experts are responsible for handling them. If the issues stem from the infrastructure, DevOps submits requests to the operations group for vertical expansion or to allocate resources for horizontal expansion as needed.

Continuous Deployment (CD)

Continuous deployment consists of the following two processes:

- Deploying to the staging environment
- Deploying to customer-facing repositories

The availability of this staging environment and of the customer-facing repositories is supported by operations teams such as infrastructure and monitoring teams, which guarantee the service availability in compliance with the Service Level Agreements (SLAs).

Let's discuss each of these processes a little bit more in detail.

Deploying to the Staging Environment

When validating a release candidate or a release, quality assurance (QA) professionals are responsible for executing Functional Tests and User Acceptance Tests (UATs). These tests can be time-consuming and often involve manual steps. An additional challenge arises during UAT, as customer QA professionals are typically involved, necessitating that the software being tested is remotely accessible, often through a VPN.

To address these requirements, automated testing facilities usually provide an option to deploy to the staging environment, sometimes referred to as the QA environment. Unlike the ephemeral environments used in continuous integration (CI), the architecture-specific platforms deployed to the staging environment remain operational after the completion of unit tests and integration tests. This allows QA professionals to conduct longer-lasting and manual tests without the pressure of quickly freeing up resources.

The deployment automation leverages much of the same codebase used to spin up the ephemeral systems and workspaces in CI, ensuring that necessary integration services – such as databases and message queues – are automatically provisioned and configured. Depending on business needs, available resources, and budget constraints, the staging environment can be scaled to accommodate the load of multiple independent integration services for various applications running in parallel, including different versions of the same application. This setup provides QA professionals with sufficient time to conduct their tests without the urgency to release resources for testing other applications.

Table 33-3 depicts the specific tasks of each professional in relation to the staging environment.

Table 33-3. *Professionals and tasks in relation to the usage of the staging environment*

Professionals	Tasks
Quality Assurance	Run on-demand integration tests as well as additional testing including functional, regression, and performance tests, to validate the application meets the quality gates required by release candidates or releases. QA professionals may also facilitate User Acceptance Testing (UAT), where real users evaluate the application to ensure it meets their needs and expectations before it goes live.
DevOps	Run the deployment process to the staging environment, ensuring that QA professionals receive a fully functional environment ready for testing. They maintain the automated testing facilities and extensions that facilitate the automatic deployment of the integration services necessary for running tests. Additionally, DevOps monitors performance metrics to ensure that they align with the expectations outlined in the Organizational Level Agreement (OLA). In the event that anomalies are detected, DevOps professionals are responsible for addressing these issues. If the anomalies are related to the platform, DevOps will raise requests with the operations team for vertical scaling or to provision the necessary resources for horizontal scaling as required.

Deploying to Customer-Facing Repositories

This process is triggered when a release candidate or a release has been created. It is typically implemented as an option that pulls the successfully tested release or release candidate package from the testing repository and pushes it to the customer-facing repository.

Note Release and release candidate packages are generated only after the codebase has passed all quality gates and tests and after completing the changelog, release notes, and documentation. During the building of a release, the packaging process typically involves digitally signing the package before pushing it to the testing repository to prevent the promotion of tampered packages to production.

Table 33-4 depicts the specific tasks of each role in relation to the staging environment.

Table 33-4. *Roles and tasks in relation to deploying to customer-facing repositories*

Role	Tasks
Product Owner	Holds primary responsibility for approving releases and release candidates, ensuring they align with business objectives and stakeholder expectations
DevOps	Maintains the automations for deploying to the customer-facing repositories and runs the actual process when necessary

Continuous Delivery

Continuous delivery is a software development practice that enables frequent and reliable deployment of tested products to production environments.

It is often implemented in the context of software delivered as a service (SaaS), since it facilitates quick and automated deployment of new releases to production repositories using automation tools that not only deploy the software and run necessary sanity checks to ensure operational integrity, but also implement appropriate release strategies to minimize or prevent service outages.

Examples of strategies are

- **Blue-green deployments**: It involves maintaining two identical environments, referred to as `blue` and `green`. The new version of the application is deployed to the idle environment, where it undergoes sanity checks. If the checks pass, traffic is switched from the current environment to the new one, performing a quick switch that minimizes downtime and risk.
- **Canary releases**: In this approach, a small number of replicas of the application are updated and deployed offline. After running sanity checks, if the new version is deemed stable, it is added to the online pool, replacing an equal number of current replicas. This process continues until all replicas are updated.

A properly implemented deployment automation, depending on the use case, must also be able to abort and exit failing before going live with a disruptive change or even better roll back the deployment if anything unexpected or wrong happens. In addition to that, it must be idempotent, meaning that re-running the deployment process multiple times does not alter the overall outcome.

Continuous delivery leverages a VCS such as Git: Git repositories contain not only the source code of the automation processes, but also the necessary configuration files and templates used to configure the deployed applications.

These automated processes can be triggered by CD management suites such as Argo CD, workflow management applications such as Argo Workflow, or generic automation servers such as Jenkins.

In the last few years, a discipline called GitOpts started spreading. It is implemented by connecting runners to Git repositories, configuring action workflows that are triggered on specific events, such as completing a pull request merging the changes into a branch. This avoids the need for a dedicated deployment tool for triggering and running the deployment.

Note It is also worth mentioning that while GitOpts can simplify deployment processes, many organizations still use specialized tools (like Kubernetes operators, Argo CD, or Flux) to facilitate GitOpts workflows.

Continuous delivery is typically managed by DevOps professionals, who implement and maintain the automation processes and run deployments as needed.

However, if continuous delivery is implemented without proper governance, it is subject to several risks, such as deploying software instances without sufficient licenses, overloading hardware resources, running conflicting deployments simultaneously, and so on.

To mitigate these risks, it is crucial to enforce change management practices, which impose for each deployment to raise a request subject to the change manager's approval. To enable a more accurate risk evaluation, these requests must also include the intended deployment schedule.

The change manager performs essential checks, such as

- Ensuring there are no conflicts with other deployments
- Verifying that sufficient resources and licenses are available
- Conducting procedural checks, such as confirming that backups will be taken and that a rollback procedure exists
- Ensuring that all involved parties are informed of any potential service outages

If all checks succeed, the request is approved, and the deployment can proceed.

If any issues are identified, the request is rejected, and necessary corrections must be made before submitting a new request.

In summary, integrating change management with continuous delivery is vital for ensuring successful deployments while minimizing risks and maintaining compliance.

Summary

This chapter provided a comprehensive overview of the software toolchain, which is an essential skill for any DevOps and DevSecOps professional. We explored the intricacies of the CI/CD toolchain, offering a thorough explanation of the continuous integration, continuous deployment, and continuous delivery processes. Additionally, we examined the shared responsibility model and emphasized the significance of implementing change management within the continuous delivery process, highlighting the risks it mitigates.

For DevOps and DevSecOps professionals, a deep understanding of the CI/CD toolchain is vital, as it forms the foundation of the entire software development lifecycle. Familiarity with the structure of these components enables professionals to streamline workflows, enhance collaboration, and improve overall efficiency. The shared responsibility model underscores the importance of collective ownership of quality and security, fostering a culture of accountability and continuous improvement that is crucial in today's fast-paced development environments.

By mastering these concepts, professionals can effectively manage risks, adapt to changes, and deliver high-quality software that meets user expectations and aligns with business goals.

CHAPTER 34

Working with Gitea Actions Workflows

After installing Gitea and having acquainted us with CI/CD concepts, it's now time to have a walkthrough of its built-in CI features: in this chapter, we have a practical overview of running Gitea Actions workflows using Gitea Actions Runners.

The Fancymonitor project indeed contains a few pre-configured Gitea Actions workflows specifically tailored to quickly show you how the Actions workflows operate in Gitea.

This chapter contains the following sections:

- The "`Pushing Fancymonitor to Gitea`" section contains a lab which shows how to set up an Actions workflow to trigger the run of the PyTest-based unit tests by executing the `test` Makefile target.
- The "`Mocking Services`" section describes the necessary steps to set up an Actions workflow to spin up mock services using customized container images leveraging Gatehill's Imposter and use them to perform component tests.

  Be wary that due to the requirements, this example is not suitable to be run on the single container Rancher setup we deployed: the intent is just to provide a baseline you can use as a reference if you want to implement component tests with Gatehill's Imposter in Gitea.
- The "`Pull Requests`" section shows how to raise a pull request for merging a feature branch into the `main` branch. The description also includes the completion of the pull request by merging, which causes the triggering of the Actions workflow which runs the PyTest-based unit tests.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_34

- The "`Trigger the Create Release Workflow`" section shows how to use Actions workflow when creating a release, so that each time a push is received in the release branch, the Actions workflow triggers the generation of the Fancymonitor's RPM packages by running the `rpm` Makefile target and pushing them to the proper Pulp3 repository using the `push-rpm` Makefile target.

What Are Gitea Actions Workflows

Gitea Actions workflows, commonly referred to as Gitea Actions, are declarative workflows that are triggered by specific events. By configuring these workflows, users can establish a CI/CD system within Gitea, managing various tasks throughout the software development lifecycle, including building, testing, and deploying code.

Gitea Actions are designed to be compatible with GitHub Actions, sharing a similar syntax and functionality. They are based on the nektos/act project (https://github.com/nektos/act), which is a tool that allows users to run GitHub Actions locally. However, it is important to note that not all features of GitHub Actions may be supported in Gitea Actions due to platform differences.

To automatically trigger workflows, the Gitea team developed the Actions Runner. This application runs as a service, registering itself with a Gitea instance and providing a runtime environment for executing workflows by launching act instances.

Once registered with an authentication token, the Actions Runner periodically checks for events that can trigger workflows. When a relevant event is detected, it spins up a dedicated container using a suitable container image that is compatible with the steps defined in the workflow.

In addition to being available as a standalone application, the Gitea Actions Runner can also be deployed as a container image. This packaging allows for quick and easy deployment; however, it requires the container to run with privileged access, as it relies on Docker-in-Docker.

Since we have already deployed an Actions Runner in Chapter 32, we can now proceed to configure a few workflows to see it in action.

Pushing Fancymonitor to Gitea

The first lab of this chapter involves using the Fancymonitor project to test the Actions workflow that is triggered by every received push, which runs the unit tests based on PyTest.

Since we have not yet created the `fancymonitor` Git repository in Gitea, we must do that now.

Create the Fancymonitor Repository in Gitea

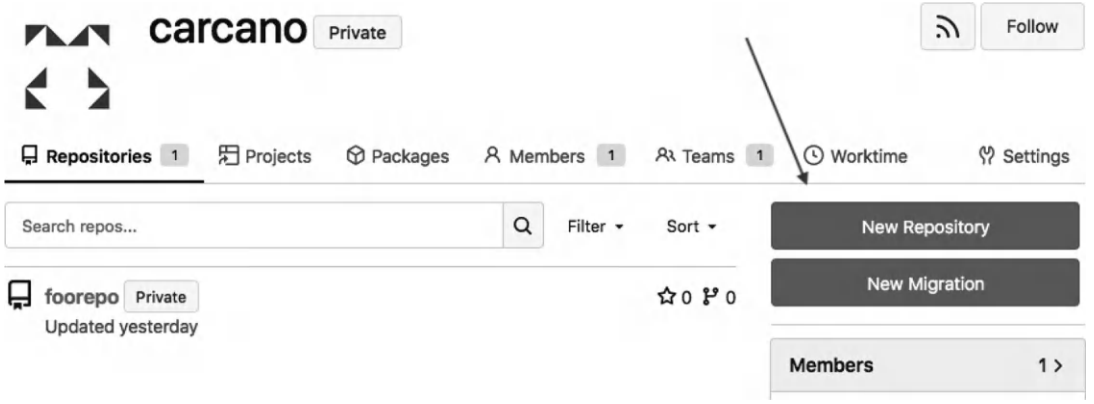
Log in to the Gitea web UI as `gitea_admin` user. Once logged in, within the `Repository / Organization` box on the top right of the page, click the `Organization` tab.

This shows the list of organizations the user has access to, as illustrated by the picture below.



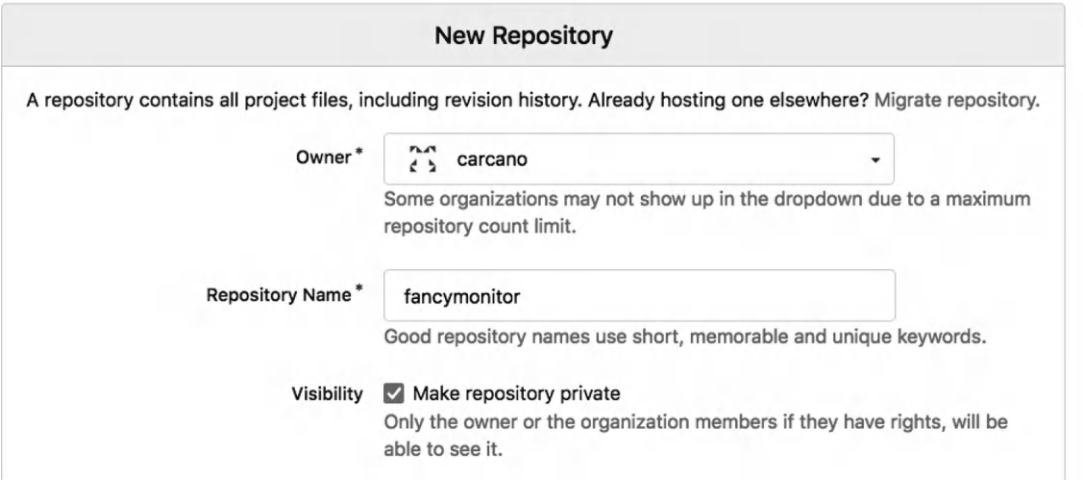
Click the `carcano` organization in the `My Organizations` list: this will show the page with all the information about the organization.

Click the `New Repository` button:

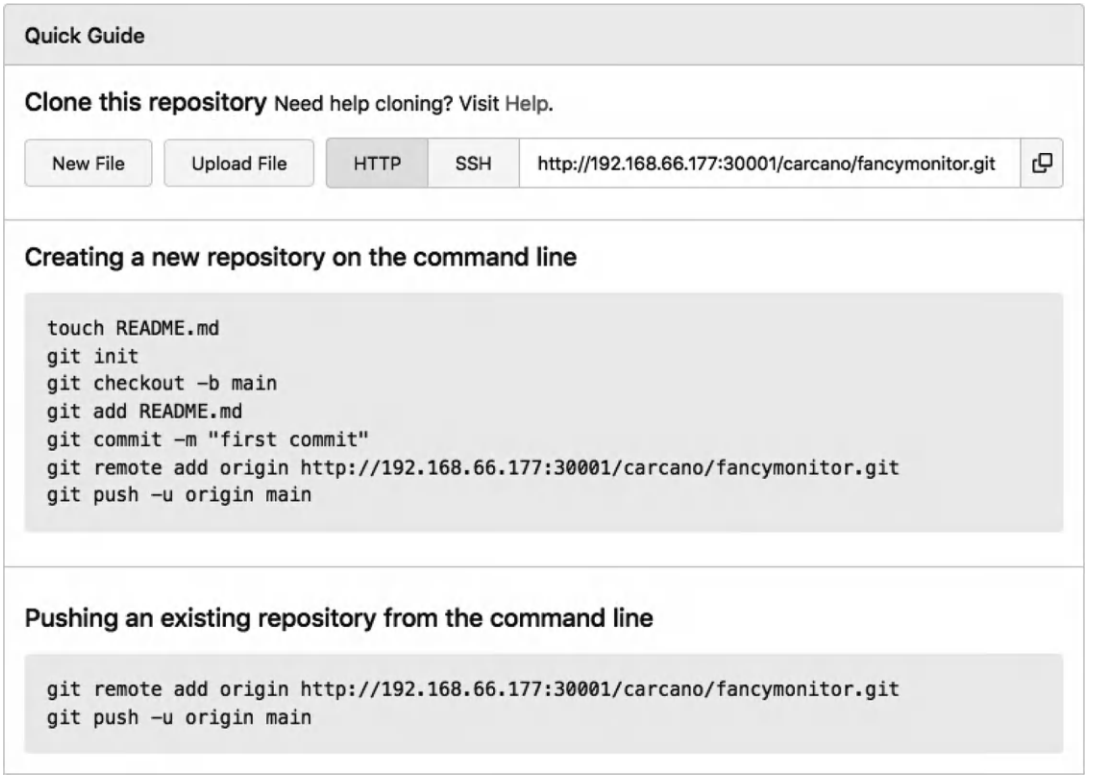


Fill in the form as shown in the below picture.

Note Since we are pushing the contents of an already existing local repository, we must NOT tick the `Initialize Repository (Adds .gitignore, License, and Readme)` option.



Once done, it displays the following `Quick Guide` with the instructions to clone or create a new Git repository and connect it to the one we have just created in Gitea.



Add the Remote Origin to Fancymonitor

Since the Fancymonitor repository already exists in the filesystem, we are in the scenario described by `Pushing an existing repository from the command line` box.

This means we only need to run the Git statement to connect the local repository to the upstream Git repository we just created in Gitea.

In Git terms, this operation is called adding a `remote`: since this is the first and the default one, it is called `origin` by convention.

Since in Chapter 22, we moved the Fancymonitor project beneath `/opt/build`, with the intention of running the CI process as the `build` user, we must switch to that user:

```
sudo su - build
```

and then change directory to `/opt/build/fancymonitor`:

```
cd /opt/build/fancymonitor
```

We can then add the `fancymonitor` Git repository we just created in Gitea as the remote origin:

```
git remote add origin \
http://192.168.66.177:30001/carcano/fancymonitor.git
```

Of course, replace the `192.168.66.177` IP address with the one suggested in the `Quick Guide`.

Pushing the Main Branch to the Remote Origin

Once the remote `origin` is added, we can set it as upstream for the `main` branch and push the branch to it by running

```
git push -u origin main
```

You will be asked for the username and password to authenticate to Gitea: for the purpose of this lab, you can use the `gitea_admin` credentials – in real-life scenarios developers obviously use their own corporate account.

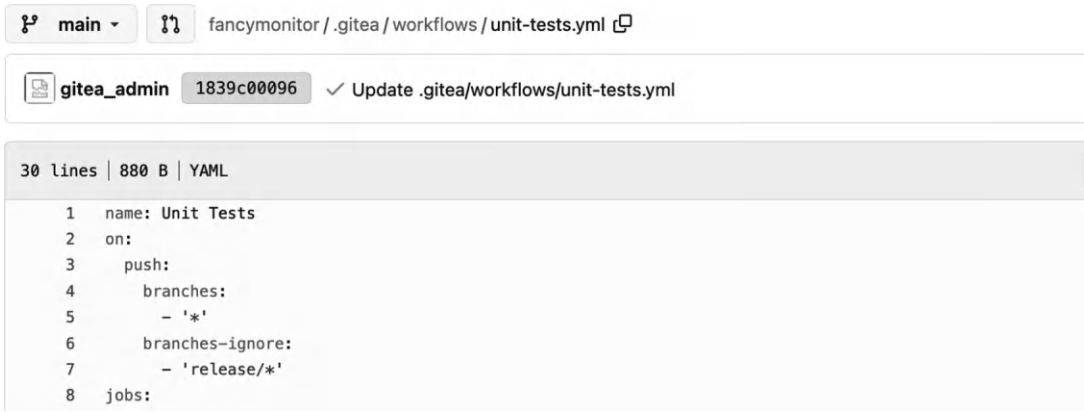
```
Username for 'http://192.168.66.177:30001': gitea_admin
Password for 'http://gitea_admin@192.168.66.177:30001':
Enumerating objects: 138, done.
Counting objects: 100% (138/138), done.
Delta compression using up to 6 threads
Compressing objects: 100% (123/123), done.
Writing objects: 100% (138/138), 26.02 KiB | 1.86 MiB/s, done.
Total 138 (delta 57), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (57/57), done.
remote: . Processing 1 references
remote: Processed 1 references in total
To http://192.168.66.177:30001/carcano/fancymonitor.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

Once the push process has completed, a copy of the `main` branch is stored into Gitea: just refresh the page in your browser to display the contents of the repository branch you just pushed.

The Unit Test Workflows

Gitea infers the Actions workflows by checking the contents of the `.gitea/workflows` directory in the `main` branch. Any found YAML file containing a workflow is then processed by Gitea, configuring the actions to be triggered.

The `main` branch we just pushed contains the `.gitea/workflows/unit-tests.yml` file: since it defines a workflow triggered at every push, Gitea immediately runs it.



Note Mind that the Actions Runner needs to pull from the Internet the proper container image, so it may take some time before it actually starts processing the workflow's jobs.

As its name suggests, this workflow runs the project's unit tests.

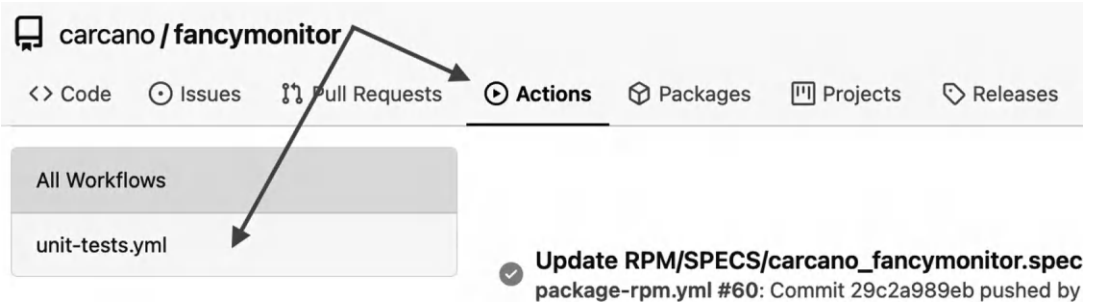
The on section (lines 2–7) configures this Actions workflow to be triggered when any branch is pushed, except if they are `releases`:

The workflow runs on a `ubuntu-latest` container (line 10), setting up inside it the workspace where to perform the following steps:

- Check out the pushed branch from the current Git repository using the `actions/checkout@v2` action (lines 12–13).
- Install the missing packages providing the library necessary to run the application and the actions contained in the workflow process (lines 14–18).

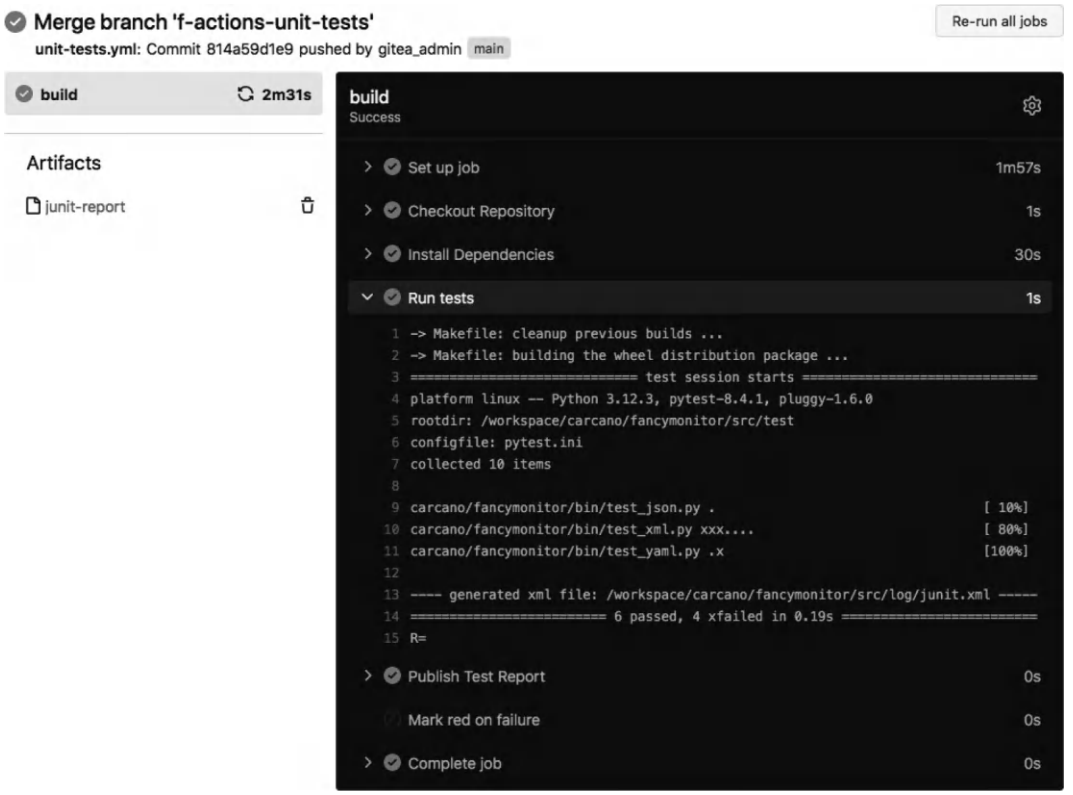
- Run the unit tests by invoking `make test` (lines 19–22). Please note how we have set `continue-on-error: true` to force the workflow to continue even if there are failed tests: this trick is necessary to make sure the next steps always run, since we want to upload the generated JUnit report as an artifact.
- Run the `actions/upload-artifact@v3.1.0` action to publish the JUnit report with the outcome of the tests into the Gitea action's artifact repository (lines 23–27).
- Since the workflow can reach this step even if there are failed tests, this last step checks the outcome of the `run_tests` step and exits returning an error condition if it was failed (lines 28–30).

We can check the outcome of the run by clicking the `Actions` button on the menu bar. Then, to display the list with only the runs related to the unit tests workflow, click `unit-test.yml` in the `All Workflows` box.



Once done, click the specific run you want to inspect to see its details: this will show you a folded step list with a colored mark to visually guess its outcome (`success`, `failure`, or `skipped`).

Note You can of course unfold each single step to inspect its detailed output: in the below picture the `Run Tests` step was unfolded to show the outcome of the unit tests reported in the console.



While running workflows you can publish any generated artifact you want to keep: artifacts published during the run are retained by default for 90 days, but you can customize the retention time by setting `ARTIFACT_RETENTION_DAYS` in the `actions` stanza in Gitea's configuration file.

If you look at the above picture, on the left, there's the list of the artifacts published during the run – in this case we have only the `junit-report` artifact.

You can then download any published artifact as necessary. In this case, for example, you can download the `junit-report` to open it with an application that can render JUnit reports in a human-readable format.

Note Besides enabling to publish artifacts attached to workflow runs, Gitea provides also a repository aimed at publishing releases of your software. If you need to do that, you can use the `release-action` action.

It is worth mentioning that, if necessary, you can manually trigger the workflow by using the `Re-run all jobs` button on the top right.

Note I strongly suggest you spend some time familiarizing yourself with Gitea's actions. The whole list is available at https://gitea.com/actions/.

Mocking Services

Note Do not run any of the commands in this section – the next practical lab is in the "`Pull Requests`" section. As anticipated, this section contains an example too complex for the stacked K3s/Rancher/Gitea setup we are using. It is published in the book only to provide you with an initial guideline to follow if you need to mock services using Gatehill's Imposter. To run it you need a full-featured Kubernetes environment, with working DNS and load balancers, and rework the example to match it.

If necessary, it is possible to mock services inside Gitea Actions workflows by defining a list of service endpoints that are spun-up before processing the workflow's steps so as to be able to call them.

Mocking services is a best practice which enables writing effective component tests. By implementing component tests, you can also exercise the line of codes performing API calls: even if these tests are not making sense of the returned values, they are anyway a good way to make sure the interfaces have been properly developed in compliance with the agreed protocol formats.

As a purely didactical example, let's see how to create a container that mocks up the Petstore Swagger application (https://petstore3.swagger.io), simulating a workflow running integration test calls to it.

Create the Mock Container

The first step is creating the mock container image.

Note This process obviously requires operating as a user granted to run `docker` statements.

Open a terminal, create a workspace and change directory into it. For example:

```
mkdir ~/mock-integration-tests
cd ~/mock-integration-tests
```

Then, create the directory for storing Gatehill's Imposter's configuration files – since we are mocking the Petstore Swagger application, just call it `petstore`:

```
mkdir petstore
```

We must now download a copy of the Petstore's OpenAPI document, saving it into the justly created directory:

```
curl -o petstore/openapi.json \
https://petstore3.swagger.io/api/v3/openapi.json
```

Lastly, create the `petstore/imposter-config.yaml` Imposter's configuration file with the following contents:

```
---
plugin: openapi
specFile: openapi.json
```

Note In this example we are intentionally keeping everything at the bare minimum, but in a real-life scenario, you are supposed to tailor much more Gatehill's Imposter's configuration to provide the most realistic possible replies to the calls.

We can now create the custom container image of the `petstore` service to mock up, deriving it from Gatehill's Imposter's container image.

Create the `Dockerfile` with the following contents:

```
FROM docker.io/outofcoffee/imposter:latest
COPY petstore/ /opt/imposter/config/
```

then just build the image by running

```
docker build -t reg.t1.carcano.corp/mocks/petstore:latest.
```

Note The above statement assumes the container images registry URL where to publish the image is `reg.t1.carcano.corp`. Once the image is built, you must push it to the container images registry.

Configure the Actions Workflow

After publishing the mocked service in the container images registry, we can move forward and focus on creating the Gitea Actions workflow which spins up the customized container image as a service before starting to process the steps list.

Configure the Container Registry Credentials

Since the Actions Runner pulls the customized container image from the container images registry, if this requires authentication, it is necessary to configure the credentials to be used for pulling.

The obvious caveat is that, besides the password being confidential information, credentials vary depending on the target container images registry: for this reason we cannot hardcode them into the workflow.

The solution is to use in the workflow code only a reference to the `GIT_T1_REGISTRY_USERNAME` variable and `GIT_T1_REGISTRY_PASSWORD` secret to be used during the pull.

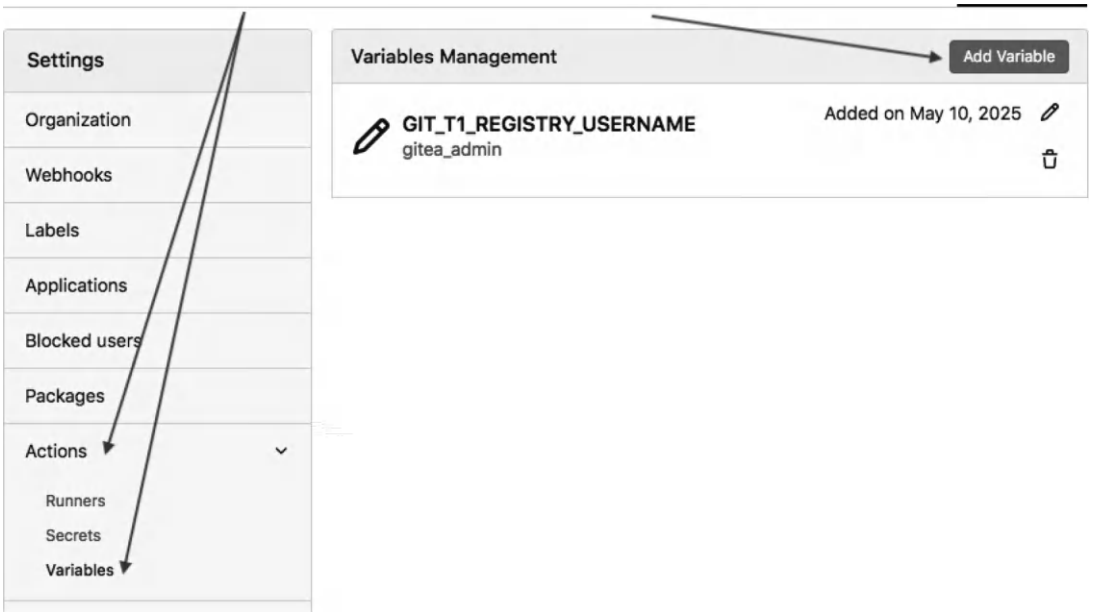
Variables and secrets can be created either at the `Repository` or `Organization` level – in this example, we create them at the `Organization` level, to have them available in every repository belonging to the organization.

Add Organization-Level Actions Variables

To create them, within the Gitea web UI, select the organization owning the Git repository you want to create the action in and click the `settings` button on the right.

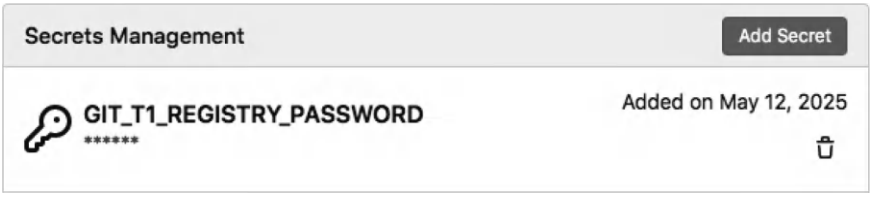
Then, from the `Settings` menu on the left, expand the `Actions` group: this reveals the links to the organization's `variables` and `secrets` – click `variables`.

Then, using the `Add Variable` button, add the `GIT_T1_REGISTRY_USERNAME` variable:



Add Organization-Level Actions Secrets

Once done, click `secrets` and, using the `Add Secret` button, add the `GIT_T1_REGISTRY_PASSWORD` secret:



Configure the Actions Workflow

Note This part is just a basic guideline – the below statements cannot work in our lab. Don't add it to the `fancymonitor` project or it will break everything.

We can finally create the Gitea Actions workflow in the `.gitea/workflows` directory as usual.

This is just a very simple model you can rework and expand to fit your use case, adding services and steps as necessary: in a real-life scenario, you would need multiple services and steps.

Here is this example's snippet:

```
on:
  push:
    branches:
      - '*'
    branches-ignore:
      - 'release/*'
jobs:
  unit-tests:
    runs-on: ubuntu-latest
    services:
      imposter-petstore:
        image: reg.t1.carcano.corp/mocks/petstore:latest
        credentials:
          username: ${{ vars.GIT_T1_REGISTRY_USERNAME }}
          password: ${{ secrets.GIT_T1_REGISTRY_PASSWORD }}
    steps:
      - name: Get First Pet
        run: |
          sleep 10; curl http://imposter-petstore:8080/api/v3/pet/1
```

The above workflow is on purpose very trivial just to let you easily figure out the structure and syntax: it spins up the `imposter-petstore` service endpoint by pulling the container image `petstore:latest` from the `mocks` repository in the `reg.t1.carcano.corp` container images registry using the specified credentials.

It then runs only a single step: sleeping 10 seconds to give the `imposter-petstore` service enough time for booting, and then it calls it using curl.

Note As we said, this is just a basic example to show you from where to start: in a real-life scenario, you must instead implement several other things, such as a loop retrying calling the endpoint for a number of retries until Gatehill's Imposter service comes up.

Pull Requests

A pull request is just a request to pull the changes from one branch into another one: it is a very common process during software development, used to better track the merges and to add an approval stage before merging as necessary.

Note In Gitea, pull request approval policies can be enforced on a per-target-branch basis when defining branch protections in the repository's settings page.

Pull Request's Merging Strategies

Gitea supports several types of pull request:

- **Create merge commit**
- **Rebase then fast-forward**
- **Rebase then create merge commit**
- **Create squash commit**
- **Fast-forward only**

Choosing the right option depends on your project's workflow preferences and how you want to manage commit history.

Note Since they obviously rely on Git merging strategies, we won't delve into their differences, postponing it to the next chapter which is entirely dedicated to Git.

Pushing a Branch

As anticipated, the `f-push-rpm` branch of the Fancymonitor project has intentionally not been merged into the `main` branch, so that we can use it to see how to manage a pull request.

To see it in action, we must first push the `f-push-rpm` branch to the remote `origin` repository.

On a terminal, as the `build` user, change to the Fancymonitor's project's root directory as follows:

```
cd /opt/build/fancymonitor
```

then, run the following statement to push the `f-push-rpm` branch to Gitea:

```
git push -u origin f-push-rpm
```

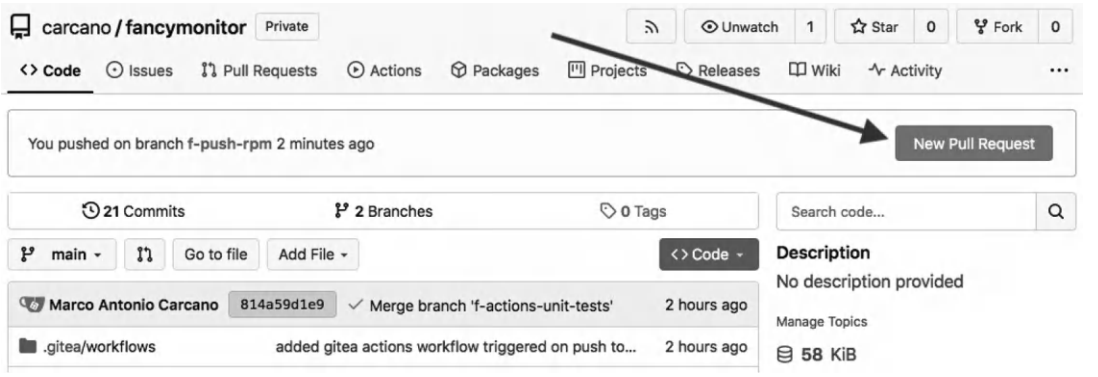
As you should guess, this operation will cause the unit-tests Actions workflow to run again – check it in the Gitea web UI, and wait for it to complete.

Raising a Pull Request

We can now raise the pull request: click `Pull Requests` on the repository's menu bar (right above the number of commits), then click the `New Pull Request` button.

It is worth mentioning that Gitea has a nice feature which shows a notification about your most recent push, along with a button to create a pull request if necessary.

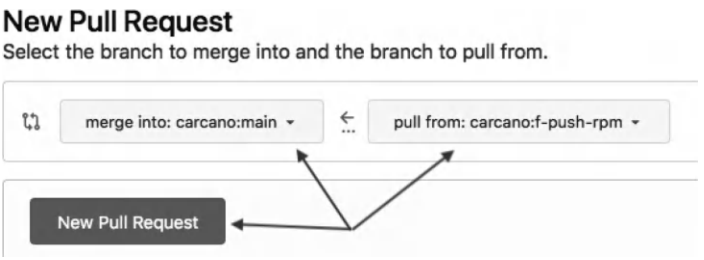
This is shown in the following picture.



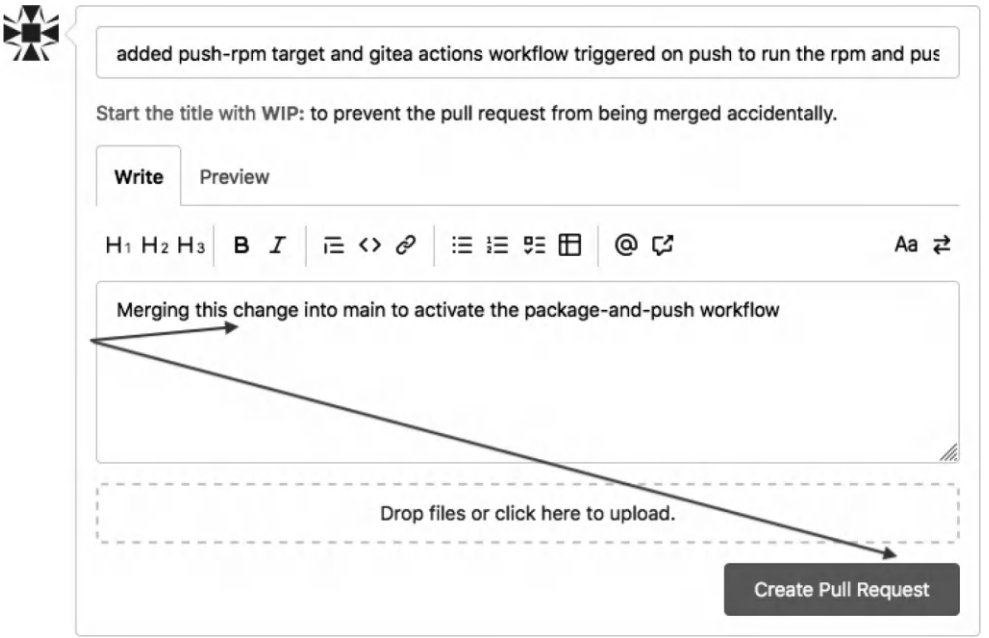
We want to create a pull request for merging the changes from the `f-push-rpm` branch into the `main` branch.

Fill in the pull request's details: the combo box on the left is the target branch (`main`), whereas the one on the right is the source branch (`f-push-rpm`).

Once done, click the `New Pull Request` button: the process moves forward showing the below form with the pull request's description.



Just fill in the rationale you are raising the pull request for – for example:



Once done, create the pull request by clicking the `Create Pull Request` button.

Note This completes the creation of the pull request (PR). Gitea now displays the PR status, which includes checks to ensure that the merge can be performed successfully. These checks not only verify the absence of merge conflicts but also assess the outcomes of any configured Actions workflows. As a result, a pull request will not be mergeable until all configured Actions workflows have completed successfully.

Completing a Pull Request

In a normal CI process, a raised pull request, even without merging conflicts, is blocked until it does not get approved by someone entitled with approval rights: merging changes into a long lasting branch, such as `devel`, any release candidate or any release branch is always a risky task, since there are real chances of breaking something used by lots of people, severely affecting their productivity.

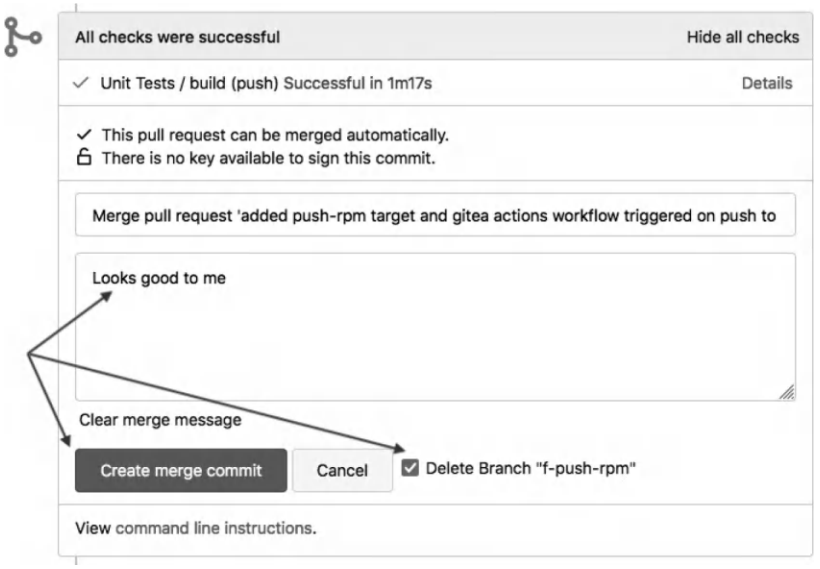
For this reason, the best practice is always to add an approval stage when merging into a long-lasting branch. The approval is typically given if the code in the pulled branch meets the project's agreed quality gates (checks are typically triggered using a dedicated Actions workflow which triggers the scan at each push).

Note Since this is a lab, and we are playing alone, we are not setting up the approval stage. Setting it up anyway is quite trivial, so we won't dig into it.

To complete the pull request, click the `Create merge commit` button:



This will show you the form for completing the pull request: write a short message to prove you are doing this only after a successful completion of the pull request review process, and, since this is a short-living feature branch, tick the `Delete branch "f-push-rpm"` option. Once done, click the `Create merge commit` button:



As you should expect, the completion of the merge process will trigger another run of the `unit-tests` Actions workflow.

Note The completion of this merge adds the `package-and-push.yml` Actions workflow to the `main` branch, which is a requisite for running the "`Triggering the Create Release Workflow`" section.

Creating a Release

A very typical workflow is the one used to package an application and push the generated packages to the proper repository, for example, when cutting a release or a release candidate.

In such a scenario, instead of ending after running the tests, the workflow continues, packaging the application and pushing it to the proper registry.

Note In this lab we are pushing to Pulp3: this is a purely didactical choice to show a few examples of integrating it into a CI/CD. It is anyway possible to rework the examples to push to repositories managed by any other repository manager, even pushing to Gitea itself: Gitea supports publishing various kinds of artifacts – for more information: https://docs.gitea.com/usage/packages/overview.

In Chapter 29, we deployed Pulp3 and created the `carcano-test-el9-x86_64` and `carcano-test-el9-aarch64` RPM repositories: in this lab, we will see how to push to Pulp3 using an Actions workflow.

Configuring the Push Target

As we already said in the previous chapter when describing CI/CD, the best practice is to have most of the automation statements directly managed by the build automation tools.

Since we are using GNU Make, the `push` process is implemented by the `push` target in the `Makefile`.

The Actions workflow just invokes it after setting the proper variables or using the proper parameters.

The `f-push-rpm` branch contains the changes to the `Makefile` implementing the push.

Let's have a look at it from the local Fancymonitor repo.

Switch to the `build` user using `sudo`, then change to the Fancymonitor's project's root directory as follows:

```
cd /opt/build/fancymonitor
```

then check out the `f-push-rpm` branch:

```
git checkout f-push-rpm
```

The changed files can be listed by simply running

```
git diff --name-only f-gpg-sign
```

This reveals that only two files contain changes – let's see the actual changes:

```
git diff f-gpg-sign -- src/Makefile
git diff f-gpg-sign -- Makefile
```

As you can see, in both `src/Makefile` and `Makefile`, we only added the `push-rpm` target.

As the target's name suggests, it pushes the generated RPM into the Pulp3 repository specified by the `REPO` environment variable.

Let's try pushing it to the `carcano-test-el9` repository matching the architecture of your system.

First, let's set the necessary variables:

```
export RELEASE=0.1.0
export REPO=carcano-test-el9-$(uname -m)
```

Then, we must build the RPM packages:

```
make rpm
```

Lastly, we can push the generated RPM packages to the Pulp3 repository:

```
make push-rpm
```

Note The `make push-rpm` target requires you to have already configured the Pulp3 client for the `build` user – we did it already in Chapter 29.

At each push, Pulp3 replies with a JSON document with information about the publishing processes' outcome of the pushed RPM file.

In general, returning a JSON document with the execution outcome is the typical behavior of every `pulp` command-line statement: you can obviously focus on specific information by piping the output to `jq`.

For example, to list all of the RPM packages published in the repository, avoiding getting all the JSON details for each of them, we can use JQ to filter only the `location_href` attribute as follows:

```
pulp rpm repository content list --repository ${REPO} \
|  jq 'map ( .location_href)'
```

The output is

```
[
  "python3-carcano-fancymonitor-1.0.0-1.noarch.rpm",
  "carcano-fancymonitor-1.0.0-1.noarch.rpm"
]
```

Note Spend some time learning how to use the `pulp` command-line tool: being familiar with it enables you to easily write automation scripts for the most common tasks. The official documentation is available at https://pulpproject.org/pulp-cli/.

We can now exit the sudoed shell by typing `exit` or pressing `CTRL+D`.

Configuring the Release Workflow

Getting back to the Gitea Actions workflow, as we saw, the other file contained in this branch is `package-and-push.yml`: it contains the Actions workflow that is triggered only when a release branch is pushed.

When started, this workflow configures on the fly the Pulp3 client inside the container and then invokes the `push-rpm` Make target.

Since when we raised the pull request we merged this branch in the main branch, this Actions workflow is already enabled, but it is still not effective: the container running it does not indeed contain the Pulp3 settings file with the information for pushing to it.

To address this problem, the Actions workflow contains a step for creating the Pulp3 settings file on the fly by using the `PULP_T1_URL`, `PULP_T1_PUSH_USER`, and `PULP_T1_PUSH_PASSWORD` variables.

We must create them at the organization level – the process is the same as described in the "`Mocking Services`" section.

Within the Gitea web UI, create the following organization-level Actions variables:

- `PULP_T1_URL`: The URL to the Pulp3 endpoint – for example, in my lab, it is `http://192.168.66.177:8080`, where `192.168.66.177` is the IP address of the host running Rancher

- `PULP_T1_PUSH_USER`: The username for authenticating the Pulp3 repository – it must match the one you used when configuring Pulp3 – the default is `admin`

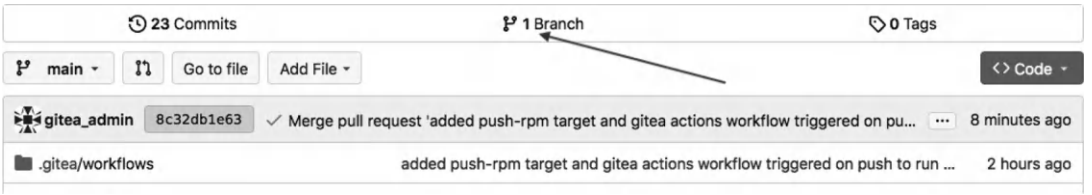
Create also the `PULP_T1_PUSH_PASSWORD` organization-level Actions secret with the user's password of the user used to authenticate to the Pulp3 registry – it must match the one you used when you configured Pulp3 in Chapter 29 (the default is `password`).

Triggering the Create Release Workflow

We have finally completed the configuration of the `package-and-push.yml` Actions workflow.

Since it is configured to be triggered by any push event to release branches (`release/*`), to see it in action, let's create the `release/1.0.0` branch from the main `branch`.

From the initial page of the `fancymonitor` repository, just click the number of branches as shown in the below picture:

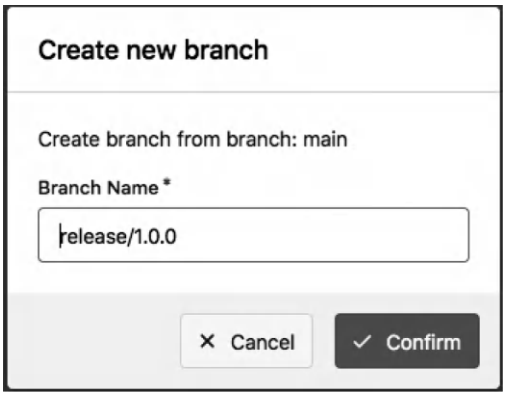


This action shows the list of all the available branches.

Then, create the `release/1.0.0` branch derived from the `main` branch – click the create new branch icon as shown below:



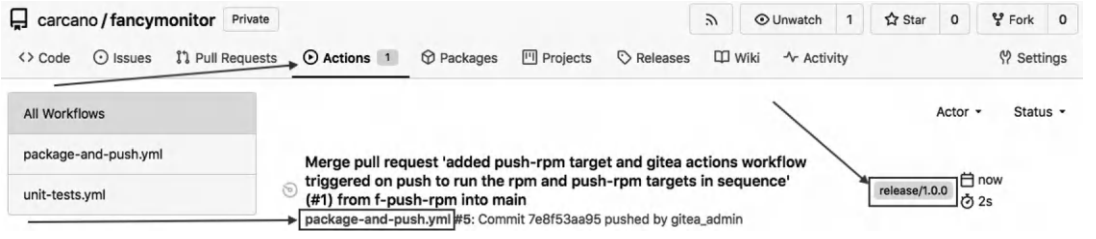
In the create new branch dialog, enter the new branch's name as shown in the below box, then click the `Confirm` button:



The `release/1.0.0` branch is created, causing the immediate triggering of the `package-and-push.yml` Actions workflow.

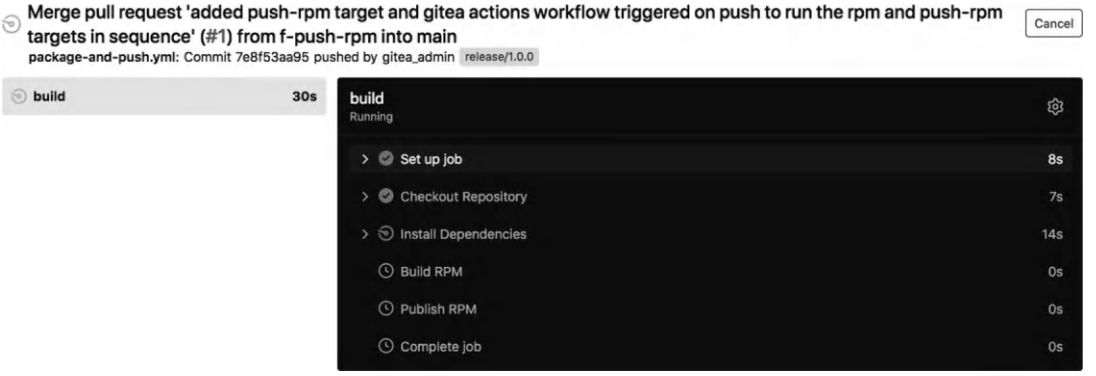
You can monitor its execution by clicking the `Actions` button of the menu bar. Then, to display the list with only the runs related to this workflow, click `package-and-push.yml` in the `All Workflows` box.

The outcome is as shown in the below picture:

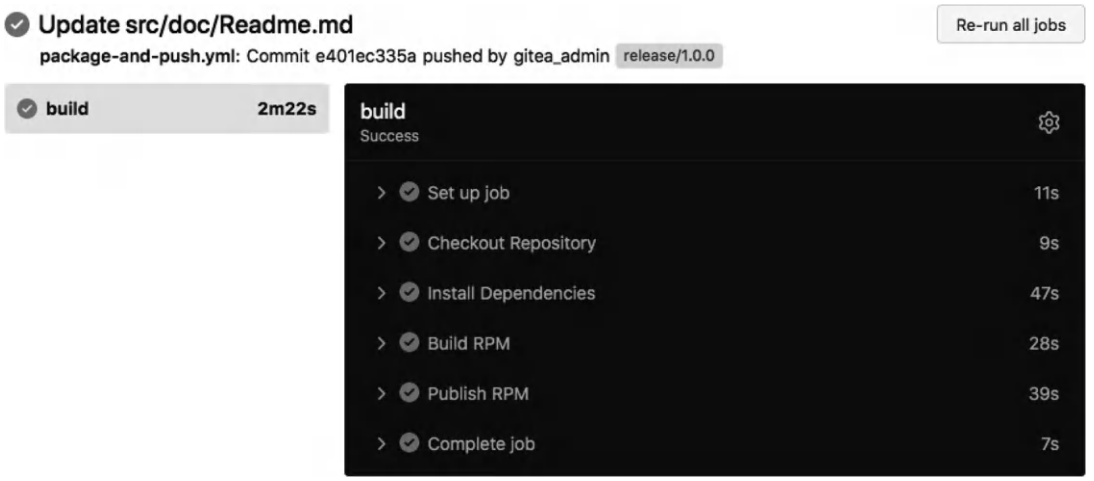


Once done, click this run to inspect the execution details.

The completion of each single stage is displayed and shown by the following picture:

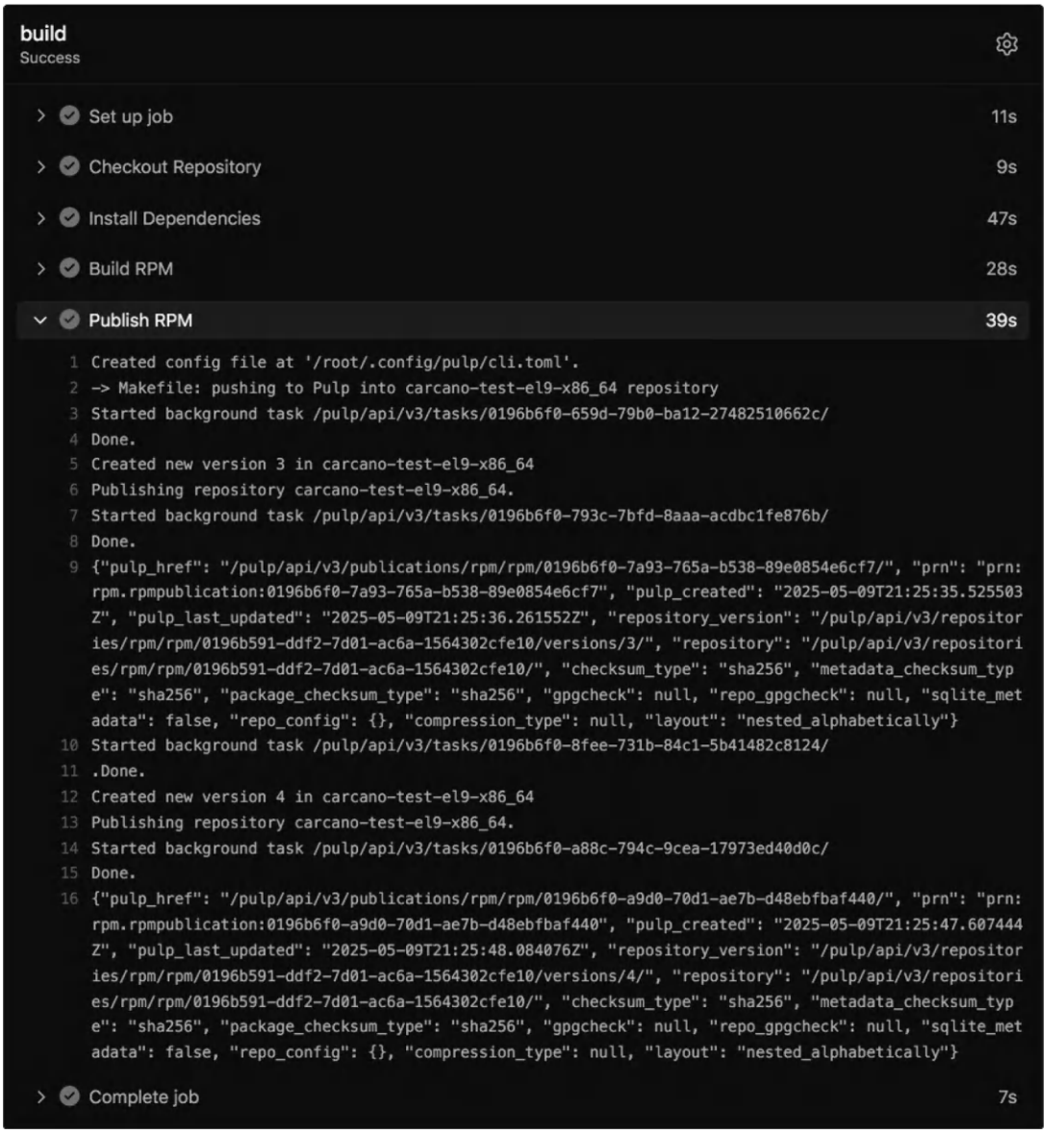


Once completed, a successful run of the workflow must look as follows:



We can of course drill down each single step to see the detailed output.

For example, to see the Pulp3 JSON replies after the push of each package, just expand the `Publish RPM` stage as shown in the following picture:



Summary

This chapter provided an overview of the potential of Gitea Actions workflows. You had the opportunity to engage in hands-on labs demonstrating how these workflows can run Makefile targets, execute unit tests, and automate the generation of RPM packages for publication in a Pulp3 repository. We explored the ability to finely control the triggering

of Actions workflows, limiting them to specific branches, such as release branches. This capability allows for the definition of tailored workflows for various use cases, ensuring actions are executed only when necessary. Additionally, we discussed integrating mock services within Actions workflows to facilitate component testing.

For DevOps and DevSecOps professionals, mastering Gitea Actions workflows is essential for automating development and deployment processes. By utilizing these workflows, teams can enhance code quality, streamline release processes, and reduce manual errors. The targeted triggering of workflows improves efficiency and resource management, while the integration of mock services supports a comprehensive testing strategy.

Overall, proficiency in Gitea Actions workflows empowers professionals to enhance automation, improve collaboration, and uphold high software quality standards throughout the development lifecycle.

PART XIII

Version Control

Within modern automated infrastructures, Version Control Systems (VCS) constitute the auditable registry of truth for the software lifecycle. DevOps and DevSecOps practitioners must possess deep structural control over these architectures to provide advanced technical escalation support to development and QA teams. Beyond mechanics, this discipline demands absolute internal rigor: professionals must reject undocumented, high-granularity checkpointing antipatterns, ensuring their own contributions function as strictly atomic, semantically defined transitions of state. Every commit must explicitly document the precise structural milestone achieved, rather than serving as an arbitrary, reactive snapshot. Consequently, this section delivers the technical framework required to govern complex branching models, resolve destructive code conflicts, and orchestrate secure merge strategies to sustain contemporary automation frameworks.

CHAPTER 35

Version Management with Git

This chapter is entirely focused on one of the cornerstones of DevOps and DevSecOps: the Git SCM.

Skills on Git are mandatory for professionals: Git is integrated in nearly every modern CI/CD solution, and Git skills are mandatory not only to effectively support development teams and QA teams, but also to maintain your own automation scripts in a collaborative environment.

Being involved in activities such as code reviews requires having a strong knowledge of branching models, merging strategies, tagging, managing pull requests, and so on.

For a professional, mastering these Git concepts is crucial. As the bridge between development, security, and operations, you will often need to collaborate with cross-functional teams, manage code changes, and ensure the integrity of the software development lifecycle.

Proficiency in Git not only enhances your ability to track changes and resolve conflicts but also plays a vital role in maintaining security and compliance throughout the development process.

Understanding these tools and techniques empowers you to implement best practices, streamline workflows, and ultimately deliver more secure and reliable software solutions.

In this chapter, we delve into operating with Git, experimenting with practical examples of all the use cases you are likely to find in your daily work as a professional.

Note Before going any further, we must create this **chapter's workspace directory** and immediately change to it. Just run: `mkdir ~/devsecops-pillars/ch35; cd ~/devsecops-pillars/ch35`.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_35

Acquainting to Git

Git is a modern Version Control System (VCS) tool that superseded Subversion (SVN) and the very old Concurrent Versions System (CVS): it is far more efficient and resilient, and one of its most appreciated features is certainly that it is a Distributed Version Control System.

It has been designed by Linus Torvalds himself, inspired by BitKeeper, … this is one of the most interesting pages of the free software history: Linus Torvalds thought that, because of the growth of the project, the VCS that they used to develop the Linux kernel was no longer suitable. He searched a lot and found that BitKeeper was the only suitable VCS – rumors said that Larry McVoy, the creator of BitKeeper, specifically designed it in the hope to meet Torvalds's needs. In addition to that, since they also offered it for free to open-source projects, the kernel developers adopted it in 2002. They used it for almost three years, until Andrew Tridgell – the father of Samba – started to develop an open-source client of BitKeeper. There were other "incidents" with the open-source community in the past, but this apparently was the last straw: BitMover rescinded the offer of free usage of BitKeeper for open-source projects. Despite Torvalds lashing out at Tridgell after this incident, this was not completely bad for the open-source world, since this pushed Torvalds to design and develop a great open-source VCS: Git.

Simplifying the workflow at its bare minimum, assuming you are operating on an already existent repository, working with GIT consists of the following list of actions:

- Pull the most current copy from the remote repository.
- Merge the pulled copy with the local one, resolving conflicts that may arise.
- Optionally check the status of files in the local repository.
- Add the modified local files to the staging area.
- Commit the modifications.
- Push to the shared repository.

These tasks are accomplished by running a list of commands such as:

```
git pull
git status
git add --all
git commit -m "a meaningful description"
git push
```

Note Mind that Git is a VCS tool specifically designed to handle source files, which are typically text files. Although its Large File Support (LFS) extension enables it to handle binary files, storing them in Git repositories can lead to a significant increase in repository size, as Git cannot manage delta changes for binaries. This severely affects pull performance and impacts the overall CI/CD efficiency. Similarly, base64-encoded binary data, such as certificates, suffers from the same limitations regarding delta management. So, the best practice is, unless there is a compelling reason to version these files with Git, use specialized credential vaults for better management.

Git Concepts

A Git repository is a directory tree whose contents are under the control of the GIT Version Control System.

Areas

A Git repository has three areas as shown in Table 35-1.

Table 35-1. *Git Concepts: Areas*

Area	Tasks
Working Area	In this area, there are: • New untracked contents • Contents modified since the last commit or since they were added to the staging area • Contents deleted since the last commit or since they were added to the staging area You can see its contents by typing `git status` Untracked contents will be listed beneath `Untracked files`.

(*continued*)

Table 35-1. (*continued*)

Area	Tasks
Staging Area (Sometimes Called Index)	In this area, there are contents that were on the working area and that have been explicitly set to be tracked using the `git add` command. Simply put, the staging area is the spot where the contents ready to be committed are. You can see these contents by typing `git status` They are listed beneath `Changes to be committed`.
Repository Area	In this area, there are only contents tracked by Git that have already been committed. Committing is the action of permanently storing contents in the repository area, adding also a comment message describing the committed change. At each commit Git computes the hash of the modified contents (including the commit message) and uses it as an identifier to locate it inside the log of the changes. Note that the GIT project has decided to replace SHA1 with SHA256, although, at the time of writing this book, the current version is still using SHA1. You can print the log of the commits of the current checked-out branch by typing `git log`

Branches

A branch in Git is a movable pointer to a specific commit, which is identified by its hash, allowing for parallel development and version control. When you initialize a new Git repository, a default branch is automatically created.

Note Mind that the term `master` came from Bitkeeper, who referred to the always true copy of the repository as the `master repository` and to the other copies as `slave repositories`. Anyway, since the community asked for a more descriptive name, Git developers, since the `2.28` version, have added the `init.defaultBranch` setting to enable specifying an alternate name for the default branch. Some Git implementations have already started to call `main` the default branch.

At the first commit time, Git creates in the `.git/ref/heads` directory a file with the same name as the current branch (e.g., `.git/ref/heads/main`).

This file is used to store the hash of the latest commit in that branch: each time you commit changes, the contents of this file are updated to reflect the new commit hash.

Note Mind that branches are bound to the repository area only. This means that although when switching branches the files of the repository area get extracted into the directory tree, the uncommitted contents already in the directory tree are retained. This situation can lead to conflicts or unexpected behavior because the uncommitted changes may not be compatible with the files being checked out from the new branch. We will see how to deal with this in detail later in this chapter when talking about stashing.

Tags

Tags are names used to conveniently identify one and only one hash: their purpose is to provide a human-readable name to refer to a hash to ease checking out contents of a given point of the story.

Install Git

Installing Git is really simple; you just need to run

```
sudo dnf install -y git
```

Configure Git

Git stores its settings in an ini-formatted configuration file.

- Global (user scope) settings are stored in the `~.gitconfig` file.
- Local (repository scope) settings are stored in the `config` file in `.git` subdirectory of the current repository.

Although you can directly edit these files with an editor, the preferred way is to manage them using `git config` statements. The advantage of using this last method is that the command line validates the statement's syntax, avoiding storing non-existent or misspelled directives inside the involved configuration file.

Note By default `git config` applies the settings to the current repository. If you want to configure global settings, you must add the `--global` option.

An example of global settings is the user's name and the email address: this information is used to identify who is the committer of each committed change, so it is very likely they are always the same across every repository. These are the very first values to set after installing Git.

To configure them just run

```
git config --global user.name 'Marco Carcano'
git config --global user.email mcarcano@carcano.corp
```

If, for any reason, you need to specify different values for a specific repository, within that repository directory tree, just run the same statement without the `--global` option.

Another option you may need to set is disabling the verification of Git's TLS server certificate: this may happen, for example, when having to deal with self-signed certificates.

If it is necessary, you can configure this exception strictly on a per-repository basis by running

```
git config sslVerify false
```

Note I won't stress you with the security implications of this: you must now be fully aware of these concepts.

Another handy setting is defining the default push method – my personal advice is to set it to `simple` as follows:

```
git config --global push.default simple
```

This configures Git to push the current branch to its corresponding upstream branch, aborting the operation if the upstream branch does not have the same name as the local branch. This is usually a wise choice, since it prevents accidental pushes to the wrong branch, ensuring that you only push changes to the intended upstream branch.

Working with Git Repositories

There are only two possible use cases when working with Git:

- Create a new repository
- Work with an existing repository

The last one is called `cloning` an existing Git repository.

Creating a New Repository

The statement to create either a local or a shared repository is `git init` followed by the name of the repository you want to create.

Local Git Repository

Creating an empty Git repository is as easy as typing `git init` followed by the name or the repository.

For example, to create the `waldo` empty repository, just type

```
git init waldo
```

Sometimes instead you have a project with existing data that you want to version control using Git.

In such a scenario you must run the `git init` statement directly inside the root of the directory tree you want to enable version control.

For example, if the `foo` is the root of the directory tree with the contents you want to version control, change directory to it and then run the `git init` statement:

```
cd foo
git init
```

When creating the repository, to store its own metadata, Git creates the `.git` directory tree inside the root directory of the versioned contents.

Shared Remote Git Repository

The straightforward limitation of a local Git repository is that you cannot share it with other people.

To overcome this, you can create a shared bare repository on a server that can be accessed also by other users.

Note A "bare" repository does not have the working area nor the staging area: it just has the repository area, so the metadata that is usually stored within the `.git` subdirectory of a personal repository is instead stored within the root directory of the bare repository. Because of this, you cannot operate using `git` statements beneath a bare repository directory: the only way to deal with it is cloning it.

When dealing with a shared bare repository, you can either clone it or configure a personal local Git repository to use it as upstream.

Bare repositories are nearly always managed using Source Code Management (SCM) software such as Gitea or Gitlab or services such as GitHub.

Anyway, for the sake of completeness, mind that the minimal requisite to create a shared bare repository is just having an SSH host with the `git` command-line tool installed on it.

As an example, supposing we have SSH access as the `git` user to the `scm.carcano.corp` host, from any client we can create the `foo` shared bare repository by running the following remote statement:

```
ssh git@scm.carcano.corp git init --bare --shared=true foo.git
```

This method works with both password-based authentication as well as public key authentication.

Note This is an extremely basic setup relying only on a single operating system level user (the `git` user). Although you can then authorize different users to connect using their public SSH key, tracing activity on setups like this is really challenging and requires you to write a lot of additional scripts.

Linking the Local Git Repository to a Shared Remote

As we already saw in the previous chapter, it is possible to configure a local repository to use a shared remote as its upstream.

We did this when we linked the local `fancymonitor` repository to the one hosted in the Gitea instance we installed on Rancher.

The statement we executed was

```
git remote add origin \
 http://192.168.66.177:30001/carcano/fancymonitor.git
```

Set the Upstream of the Current Branch

While running that Fancymonitor-related lab, we also set the upstream remote for the current branch.

While in the `main` branch, we did it by running

```
git push -u origin main
```

This command linked the local `main` branch to the corresponding branch on the `origin` remote repository. In addition to configuring the upstream link, it also performed the initial push.

By default, Git pushes a local branch to a branch with the same name in the remote repository. As a result, immediately after the push, the `main` branch of the local Git repository and the `main` branch of the remote repository will have identical contents.

Cloning an Existing Repository

The process for dealing with existing shared Git repositories is cloning them from their remote locations using the `git clone` command followed by

- The URL of the Git repository you want to clone
- Optionally, the name of the directory you want to use as the destination of the clone

Git supports a few transport protocols for cloning and synchronizing contents, as summarized in Table 35-2.

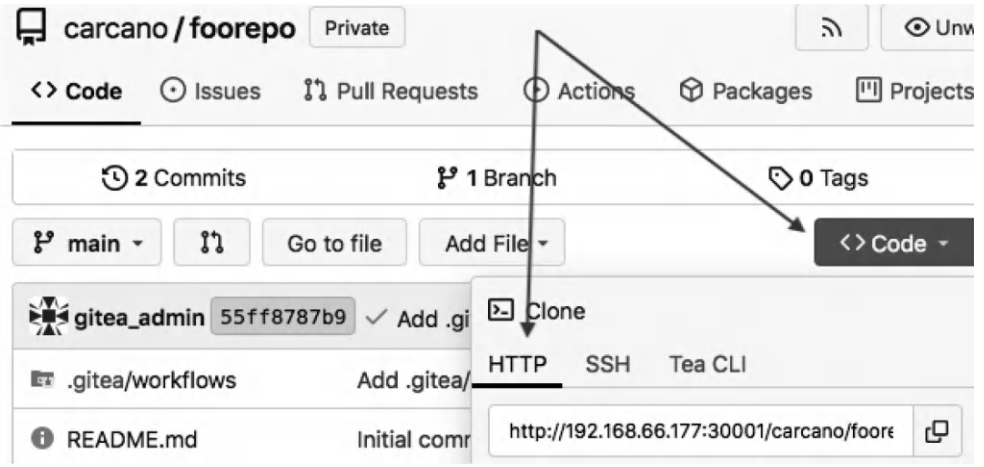
Table 35-2. *Git's transport protocols*

Protocol	Description
file://	URL of a repository stored into the local filesystem – since it is the default protocol, it can be omitted in Git clone statements.
ssh://	URL of a repository stored on the filesystem of a server that can be reached via SSH.
http:// or https://	URL of a repository that can be reached by using HTTP or HTTPS protocols.

To spare users from having to infer Git repository URLs, modern SCMs always provide a button in their Web UI to show as a hint the URL to use to clone a repository.

For example, from any repo in the Gitea web UI, you can click on the `<> Code` button to reveal the textbox with the repository URLs.

Please mind that in this lab, because of the already mentioned technical limitations caused by the stacked scenario, it was not possible to configure SSH, and so we must use the HTTP URL.



Since we are about to use it as the baseline in the next lab, from the user's home directory, let's clone the `foorepo` repository:

```
cd ~
git clone \
http://192.168.66.177:30001/carcano/foorepo.git foorepo
```

When asked for the username and password, specify `gitea_admin` along with its password – in a real-life scenario, you would instead typically use your own corporate account.

Note When you clone a remote Git repository, all its metadata is fetched, including the list of remote branches and tags. Additionally, the contents of the default branch (usually `main` or `master`) are automatically checked out into your working directory, making it the active branch in your local repository.

Version Control Operations

We can now run a short lab to use the just cloned `foorepo` repository to explore Git operations.

Let's start by changing into its root directory:

```
cd foorepo
```

Populating the Working Area

The simplest way to populate the working area is to add files and directories to the current Git repository, either by creating them or copying them from elsewhere. For example:

```
mkdir fred
cp -dR /usr/share/doc/curl .
cp -dR /usr/share/doc/git/howto .
```

Now that we have added some content, we can see how Git perceives these changes by running

```
git status
```

The output will be similar to

```
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    curl/
    howto/

nothing added to commit but untracked files present (use "git add"
to track)
```

As expected, the `curl` and `howto` directory trees are listed as `Untracked files`. Remember that everything untracked belongs to the working area.

Note You may wonder why you don't see the `fred` directory too in the list. This is because the directory does not contain any files: this means that, by the versioning perspective, there are no versionable contents.

Staging Modified Contents

As indicated by the output of `git status`, we can add contents to the staging area using the `git add` command followed by the path of the contents we want to stage. For example, to stage just the `curl/README` file, run

```
git add curl/README
```

If we run `git status` again, we will see

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   curl/README

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    curl/BUGS.md
    curl/CHANGES
    curl/FAQ
    curl/FEATURES.md
    curl/TODO
    curl/TheArtOfHttpScripting.md
    howto/
```

We can of course stage an entire directory tree. For example, to stage the whole `curl` directory tree, type

```
git add curl
```

If we run `git status` this time, we will see the full list of files in the `curl` directory in the `Changes to be committed` list, while the `howto` directory still remains in the working area.

It is of course possible to stage all the files in the working area: just specify the `--all` parameter right after the `git add` command.

So, let's stage the remaining contents:

```
git add --all
```

If we run `git status` again, this time every content of both `curl` and `howto` directory trees will be listed in the `Changes to be committed` list.

This means we have logically added every content in the working area to the staging area.

Mind that we are staging the current version of the contents; if we change anything in the working area after staging, we will have two different versions: one in the staging area and the new version in the working area.

As an example, let's modify the contents of the `curl/README` file:

```
echo "" >> curl/README
```

If we run `git status` once again, this time `curl/README` is shown twice – the one listed beneath `Changes to be committed` is the one we previously staged, while the version listed under `Changes not staged for commit` is the one we just modified.

Checking Differences

If necessary, it is possible to show the differences between the files in the working area and their latest committed version by using the `git diff` command followed by the path to the file you want to see the differences.

For example:

```
git diff curl/README
```

produces the following output:

```
diff --git a/curl/README b/curl/README
index b690816..034a89e 100644
--- a/curl/README
+++ b/curl/README
@@ -53,3 +53,4 @@ NOTICE
   Curl contains pieces of source code that is Copyright (c) 1998, 1999
   Kungliga Tekniska Högskolan. This notice is included here to comply
   with the distribution terms.
+
```

The syntax is the same as the `diff` command-line utility. In this case, the only detected change is the + character shown at the end of the output, meaning the trailing empty line of the `curl/README` file in the working area.

Note If you omit the file path and just run `git diff`, Git will display the differences for all modified files in your working area.

To compare the files you have staged (in the staging area) against their latest commit, use the `--cached` option with `git diff`.

For example, to see all staged changes compared to the latest commit, run

```
git diff --cached curl/README
```

In this case, since the `curl/README` file was never committed before, the output is the entire content of the `curl/README` file.

Note If you omit the file path and just run `git diff --cached`, Git will display the differences for all modified files in your staging area.

Let's now add this last-minute change to the staging area, just run

```
git add curl/README
```

Note You may need to exclude certain files or directories from being version controlled by Git. This can be easily accomplished by creating a `.gitignore` file that lists the items you do not want to track. Please note that the `.gitignore` file itself must be staged and committed. Since the Fancymonitor project includes a `.gitignore` file, I recommend reviewing it. You can find the official documentation with the syntax at https://git-scm.com/docs/gitignore.

Unstaging Contents

Sometimes, you may need to remove contents from the staging area (this process is often called unstaging): in such a need, you must run the `git reset` command.

If you want to remove only specific contents from the staging area, for example, only the `howto/new-command.txt` file, just provide the path to the file in the `git reset` command:

```
git reset HEAD howto/new-command.txt
```

`HEAD` refers to the commit containing the contents you want to be restored in the reset operation. In this example, it is the current commit that your Git repository is pointing to. More specifically, it represents the latest commit on the currently checked-out branch.

Mind that when resetting to the last commit, you can omit the `HEAD` word and just run

```
git reset howto/new-command.txt
```

An alternate way is using the `git restore` command with the `--staged` option. For example, to restore the `howto` directory tree, type

```
git restore --staged howto
```

Let's check the outcome running `git status`:

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   curl/BUGS.md
    new file:   curl/CHANGES
    new file:   curl/FAQ
    new file:   curl/FEATURES.md
    new file:   curl/README
    new file:   curl/TODO
    new file:   curl/TheArtOfHttpScripting.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    howto/
```

Committing Contents

When you feel the changes you made are ready to be made permanent, you can commit them to the repository along with a short descriptive message.

The command to use is `git commit` followed by `-m` parameter and the comment that describes the changes you are committing.

To move forward with our example, first we must add again all the contents to the staging area:

```
git add --all
```

Now that everything we want to commit has been staged again, we can run the following statement:

```
git commit -m "added curl's command line tool documentation"
```

Note Again, mind that commits do apply to content changes: as we have just seen, if you created an empty directory this will never be added to the staging area, and so will never be included among the committed contents.

If we now check the current status:

```
git status
```

It confirms that one commit has been added – our local `main` branch is indeed now ahead by one commit from the cloned repository:

```
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
```

Amending a Commit

It may happen that, after a commit, you realize that you forgot something or made a mistake in the comment.

For example, I forgot to mention in the comment that the added documentation includes also the one from Git.

We can sort out my mistake as follows:

```
git commit --amend -m "added curl's and git's command line tools
documentation"
```

Mind the amend feature is not limited to amending the comment's oversights or errors, but it applies to everything in the staging area when amending the commit.

For example, let's copy the directory to the working area:

```
cp -dR /usr/share/doc/jq .
```

and immediately add it to the staging area

```
git add jq
```

We can now add it to the current commit by just using again the `--amend` option:

```
git commit --amend -m "added curl's, git's and jq's command line tools
documentation"
```

If we now check the status again:

```
git status
```

As you can see, despite having committed three times so far, it still reports that our local `main` branch is still ahead by one commit from the cloned repository – this is because we didn't create new commits: we just amended the current one.

Note Amending commits pulled from or previously pushed to remote repositories is very risky: someone else may have already added another commit that has the commit you are amending as parent. In this case the correct way to go is using `git revert` instead.

Listing Changes in the Current Branch

From time to time, you will probably have a look at the current branch's list of commits. You can display it by just running

```
git log
```

The output should be similar to the following one:

```
commit 0d42baa622c309f8c0d97519cd646d34030af0f5 (HEAD -> main)
Author: Marco Antonio Carcano <marco.carcano@carcano.corp>
Date:   Sun May 25 11:57:54 2025 +0000

     added curl’s, git's and jq’s command line tools documentation

commit b41051836f3d2d3fce3418d798c98fd5daa5499a (origin/main, origin/HEAD)
Author: gitea_admin <gitea_admin@carcano.corp>
Date:   Sun May 25 11:34:23 2025 +0000

     Add .gitea/workflows/onpush.yml

commit e38fcc5305a0b66b9975e8e293bf76175a43941e
Author: gitea_admin <gitea_admin@carcano.corp>
Date:   Sun May 25 11:30:45 2025 +0000

     Initial commit
```

Reverting a Specific Commit

It may happen that you want to revert the changes of a specific commit, without affecting the contents of the other commits that have been made after it.

This is possible by using the type `git revert` followed by the hash of the commit you want to be undone.

The outcome is a new commit that removes the changes that were applied during that commit.

Note It of course can happen that the changes to undo conflicts with the current contents – in this case you will have to manually solve the conflicts.

This process is particularly effective when you are the owner of a commit pushed to the remote repository and you realize you made mistakes: as we said in such a scenario you cannot simply amend the commit, since potentially other people may have started working on it in the meantime: since `git revert` undo the changes into a new commit, then there is no risk to interfere with someone else’s activity.

For example, if you want to revert the changes that were made in the second last commit `b41051836f3d2d3fce3418d798c98fd5daa5499a` in my scenario), type

```
git revert b41051836f3d2d3fce3418d798c98fd5daa5499a
```

Note As you probably inferred, this command does not cause the removal of the commit with id `b41051836f3d2d3fce3418d798c98fd5daa5499a`.

If you check now, you will see the `.gitea/workflows/onpush.yml` has been removed. And if you run `git log` now, you will see the new commit that was used to revert the change.

Rewind to an Earlier Point in History

Sometimes, after creating several commits, we may find that some of the recent changes are no longer needed or even just wrong. In such scenarios, we need the ability to rewind the Git history to a specific commit from which we can restart our work.

To address this need, Git provides the `git reset` command, which we already introduced when talking about how to un-stage contents.

Note As anticipated, beware that resetting commits may cause issues to other collaborators that are using your commits. Unless you are certain of what you are doing, avoid using `git reset` on commits that you have already pushed to a remote repository.

The `git reset` command can have three distinct behaviors: `--mixed`, `--soft`, and `--hard`.

Table 35-3. *Git reset use cases*

Area	Tasks
`git reset --mixed <commit>`	This is the default behavior: the changes from the commits are removed from the staging area and are left in the working area, allowing you to make further modifications to those files. The typical use case is making additional changes (that maybe were forgotten early) to the files and eventually putting everything as a whole into a larger commit. We already saw this example, used to remove from the staging area the `howto/new-command.txt` file: `git reset HEAD howto/new-command.txt`
`git reset --soft <commit>`	It extracts the changes from the specified commits to the staging area. It is typically used when you want to consolidate a group of small commits into a larger one.

`git reset --hard <commit>`	As the options suggest, it performs a hard reset.
	It indeed
	• Resets everything in the staging area.
	• Resets everything in the working area.
	• Moves back the branch pointer to the specified commit – this means the commits are no longer in the current branch's history, despite being in the `reflog` for a period of time.
	
	The outcome is that we reach the state there was in the commit you specified.
	
	For example, to go back to how we were three commits ago, just type
	`git reset --hard HEAD~2`
	
	or specify the hash of the commit we want to revert to
	`git reset --hard <hash>`

Note Git automatically cleans up unreachable objects, including `reflog` entries, through a process called pruning. This occurs when you run commands like `git gc` (garbage collection) or explicitly invoke `git prune`. By default, Git retains reflog entries for commits for 90 days before they become eligible for pruning. You can configure this timeframe or manually expire `reflog` entries using `git reflog expire`.

Comparing Commits

It is possible to compare different commits to see which changes have been applied by using the `git diff` command.

We have done this several times in previous chapters while discussing the development of the Fancymonitor application.

To practice this feature again, switch to the `build` user:

```
sudo su - build
```

Then, navigate to the Fancymonitor project root directory:

```
cd /opt/build/fancymonitor
```

and check out the `f-json-config` branch:

```
git checkout f-json-config
```

To perform a comparison, we must first have a look at the Git history to see the list of previous commits:

```
git log --oneline
```

The interesting part of the output is

```
261c6c1 (HEAD -> f-json-config, origin/f-json-config) Added
8401317 Merge branch 'f-xml-config'
fd626c6 (origin/f-xml-config, f-xml-config) Created ... 737e830 Merge
branch 'f-xml-only'
81752c3 (origin/f-xml-only, f-xml-only) created the ...
```

As an example, let's compare the current branch with the `f-xml-config`, using the `--name-only` option to list only the changed files:

```
git diff --name-only f-xml-config
```

the output is

```
src/carcano/fancymonitor/__main__.py
src/carcano/fancymonitor/core/configuration.py
src/carcano/fancymonitor/loaders/json.py
src/carcano/fancymonitor/models/__init__.py
src/carcano/fancymonitor/models/alert.py
src/carcano/fancymonitor/models/config.py
src/carcano/fancymonitor/models/defaults.py
src/carcano/fancymonitor/models/notify.py
src/carcano/fancymonitor/models/probe.py
src/carcano/fancymonitor/models/statement.py
src/etc/config.json
```

Note In this case we used the branch name as the compare target, but it is of course possible to use the commit id – for example, since `fd626c6` is the last commit id of the `f-xml-config` branch, `git diff --name-only fd626c6`.

It is of course possible to view the changes file by file. For example:

```
git diff f-xml-config -- \
src/carcano/fancymonitor/core/configuration.py
```

Note Similarly, it is possible to use the commit id as the compare target also when viewing the changes file by file – for example, `git diff fd626c6 -- \ src/carcano/fancymonitor/core/configuration.py`.

Since for now we don't need to run other commands as the `build` user, let's exit the current sudoed shell:

```
exit
```

Remote Repositories

Git enables you to set a remote shared bare repository as the upstream of your local repository: this means that, whenever you want to publish the committed changes of a branch, you can push that branch to the upstream remote shared repository.

Git calls `remotes` the list of these remote repositories configured to pull from and push to the branch.

Note Mind that you can manually specify the branch to push, or you can specify a default target. If necessary, you can even push all the branches as a whole.

There are two ways to work with remote repositories: clone an existing remote or configure your local repo to use a remote as upstream.

Cloning a Remote Repository

This is the easiest way to start, since this way the remote repository is automatically configured as the upstream, and the checked out branch gets automatically configured to use it as the upstream, tracking changes from it.

We already tried this when we cloned the `foorepo` repository by running

```
git clone \
http://192.168.66.177:30001/carcano/foorepo.git foorepo
```

Note In this example, because of already explained technical constraints, we are using HTTP as the transport protocol, which is unencrypted and so does not provide confidentiality: this is acceptable, since we are in a lab, but in real life you must instead use either HTTPS or SSH as transport protocol.

When dealing with HTTP or HTTPS transport, to avoid being asked to enter both username and password while pulling or pushing, it is possible to embed the username in the cloning URL as follows:

```
git clone \
http://gitea_admin@192.168.66.177:30001/carcano/foorepo.git \
foorepo
```

This way at each synchronization operation we will be asked only for the password.

Listing Remote Repositories (Remotes)

After cloning a shared Git repository, Git assigns the word `origin` as a label to the automatically configured reference to the upstream remote repository.

To see this in action, change directory to get into the `~/foorepo` repository we previously cloned:

```
cd ~/foorepo
```

The configured remotes can be listed by running

```
git remote -v
```

the output is as follows:

```
origin     http://192.168.66.177:30001/carcano/foorepo.git (fetch)
origin     http://192.168.66.177:30001/carcano/foorepo.git (push)
```

Please note how there are two different entries of the same remote (`origin`):

- The first is used to fetch
- The second is used to push

This means that, as necessary, you can implement asymmetrical workflows by using different remotes for pushing and pulling.

So, a remote is nothing but the link to a remote shared bare Git repository used to host the remote versions of one or more local branches.

Linking to a Remote Repository

When creating your own local repository from scratch, you need to set up everything by yourself for the first time. This includes manually linking the remote origin.

We already went through this process when we configured the upstream of the `fancymonitor` repository by running

```
git remote add origin \
http://192.168.66.177:30001/carcano/fancymonitor.git
```

Note This is a GIT repository wide operation, so you don't need to repeat it every time you create a branch.

Also in this scenario, if the remote repository requires authentication, it is convenient to include the username in the URL of the clone statement. This way, since the username has been permanently set, whenever you interact with the remote you just have to type the password.

Note Mind that `origin` is the commonly assigned label to the first (and often only) linked remote repository – although you can assign any name to this alias, it is wise to follow the common convention to avoid adding to the workflow unnecessary entropy (and headaches).

Working with Branches

Very often it is needed to have multiple parallel lines of development, each one of them focusing on a particular topic of a project, that in the end gets merged into the main project: this need can be addressed by creating as many branches as many are the parallel developments.

These feature/topic related branches are often referred to as "short-living branches" or temporary, in contrast with the "long-living" or permanent branches, such as the `develop` and the `main` branches used when adopting the GitFlow branching model.

Note More information on the GitFlow branching model can be found at https://git-flow.readthedocs.io/en/latest/presentation.html.

Switching from a branch (and so from a line of development) to another is called `checkout`.

Check Out a Branch and Managing Branches

As we saw, a default branch (called `main`, although the legacy name was `master` is still broadly used) is automatically created as soon as the repository is initialized.

When necessary, we can fork other branches from the `main` branch: for example, we can fork a branch called `develop` to use this long-living branch as a collector of all the commits and merges of other branches. Each time we have a stable situation, and we think it is suitable for being permanently stored, we can merge the `develop` branch into the `main` branch pulling changes from it.

To play a little with this, switch again to the `build` user:

```
sudo su - build
```

and change directory to the Fancymonitor project root directory:

```
cd /opt/build/fancymonitor
```

As you know, the project contains several feature branches: we can list the existing local ones by typing

```
git branch -l
```

The output is as follows:

```
  f-actions-unit-tests
  f-bump-lxml-requirements
  f-gpg-sign
* f-json-config
  f-make
  f-push-rpm
  f-pytest
  f-rpm
```

```
  f-setuptools
  f-xml-config
  f-xml-only
  f-yaml-config
  main
```

As you can see, the current checked out branch (`f-json-config` in this case) is marked with a * symbol.

Creating Branches

To create a new branch, just type a `git branch` followed by the name of the branch you want to create.

For example, to create `dev` branch from the `main` branch:

```
git branch dev main
```

Note If you want to use the current checked out branch as the parent of the new branch, you can omit the last parameter with the name of the source branch (`main` in this example). For example, the command would be `git branch dev`.

This command creates the new head file `.git/refs/heads/dev` and stores into it the same commit hash of the branch you are currently working on.

The `git branch` command just creates the new branch – if you want to use it, you must first check out to it as follows:

```
git checkout dev
```

Note When creating a new branch from the current one, it is possible to condense everything in just one statement by providing the `-b` option to `git checkout`. For example, `git checkout -b dev`.

By now until the next time you check out a different branch, every commit you do gets stored beneath the list of commits of the `dev` branch. The hash of the last commit,

called the HEAD, is stored into a file with the branch name (dev in this case) beneath the `.git/refs/heads` directory. At each new commit, the contents of this file,`. git/refs/heads/dev` in this case, gets replaced.

Note As previously mentioned, when you clone a repository, the metadata containing information about all branches and tags defined on the remote server is fetched in its entirety. However, only the contents of the default branch (usually `master` or `main`) are copied and extracted. Therefore, it is essential to remember to check out the branch you want to work on before proceeding; otherwise, you will be working with the contents of the default branch.

Renaming a Branch

If you change your mind on the name of the current branch, you can modify it by using `-M` option.

For example, to change the name of the current branch (the dev branch) into `develop` type

```
git branch -M develop
```

Note The renamed branch is only the local one: remote branches cannot be renamed – the only way for renaming a remote branch is pull a local copy, rename it, push it to the remote repository and lastly delete the branch with the old name from the remote repository. Be wary that after renaming a remote branch, other collaborators will need to update their references to point to the renamed branch.

Deleting a Branch

If you are working with a short-lived branch (e.g., a branch dedicated to developing a specific feature) you can delete it right after merging back to the parent.

To delete a branch type `git branch -d` followed by the name of the branch you want to delete.

As an example, let's delete the `develop` branch: since it is the current branch, we must first check out another branch, such as the `main` branch:

```
git checkout main
```

We can now delete the `develop` branch by running

```
git branch -d develop
```

Mind that if the branch you want to delete has been partially merged because of conflicts, you must delete using the uppercase `D`. So, for example, supposing you want to delete the `f-useless-feature` branch, you must type `git branch -D f-useless-feature`.

Stashing

When checking out across different branches, if there are no conflicts, Git will successfully switch branches and carry your uncommitted changes over to the new branch. A common pitfall can arise when there are uncommitted changes conflicting with files that differ between the two branches: in such a scenario, Git will block the operation.

The only available workarounds using the commands we learnt so far are

- Resetting both the working area and staging area, with the straightforward downside of losing everything you did since the last commit
- Committing, but this may run counter to the good practice of committing only when the contents make sense overall

As an example scenario pushing you into this pitfall, suppose that, while you are in the middle of your development in a feature branch, you are asked to immediately switch to another branch to fix something urgent.

To address this need, avoiding dirtying Git's history by creating an unnecessary commit, Git provides the `stash` feature, which enables you to save into a stack the content of the staging area.

Note Mind that Git stash by default saves all tracked changes in both the staging area and the working area. If you need to include new, untracked files, you must use the `-u` command-line switch. Conversely, if you want to stash only what you have already staged, use the `--staged` flag; or use `--keep-index` if you want to stash everything but keep your staged changes active in your workspace for further work.

To see this in action, let's check out from the `main` branch a new one called `f-docs`:

```
git checkout -b f-docs main
```

To simulate a real-life scenario, we must make some changes in both the staging area and working area.

Let's start by modifying the contents of the `src/doc/README.md` file to be as follows:

```
# Fancymonitor

Official documentation
```

And add it to the staging area:

```
git add src/doc/README.md
```

Only for the sake of trying `git diff` with contents in the staging area, run

```
git diff --cached src/doc/README.md
```

This confirms we have changes in the staging area. We still need to add something to the working area:

```
touch src/doc/Changelog.md
```

We are now done with the preparation of the real-life scenario.

Let's review the current situation:

```
git status
```

The output is

```
On branch f-docs
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   src/doc/README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    src/doc/Changelog.md
```

So, as expected, the `src/doc/README.md` file is in the staging area, while `src/doc/Changelog.md` is still in the working area.

Here we begin our simulated lab for learning `git stash`: let's suppose we get asked to urgently change something in the `f-xml-config` branch.

As you can see, in the current scenario, we cannot just check out another branch, since both the staging area and working area are unclean and the outcome will be unpredictable.

Stashing Contents

So, it is exactly here where the stash command comes to help: since in our case not only the staging area, but also the working area is dirty, we must use the `-u` command-line switch we stash the contents of both:

```
git stash -u
```

The output is as follows:

```
Saved working directory and index state WIP on f-docs: 814a59d Merge branch
'f-actions-unit-tests'
```

Since now both the staging area and the working area are clean, it is possible to safely check out the `f-xml-config` branch without conflict or dirtying it with unnecessary contents:

```
git checkout f-xml-config
```

We can now do whatever we wish, adding contents and committing them or even stashing again: it is indeed possible to add contents to the stash stack as necessary, even before jumping from branch to branch.

So, for example, let's add the `src/etc/oddconf.xml` file to the staging area:

```
touch src/etc/oddconf.xml
git add src/etc/oddconf.xml
```

and stash the contents again:

```
git stash
```

This time the `-u` command-line switch is not necessary since the working area is still clean.

Listing the Stashes in the Stack

As you probably inferred, every time you stash, the stashed contents are pushed to the same stack.

We can list the stacked stashes by typing

```
git stash list
```

The important bit of the output is

```
stash@{0}: WIP on f-xml-config: fd626c6 Created ...
stash@{1}: WIP on f-docs: 814a59d Merge branch ...
```

Just mind that

- `stash@{n}` is the identifier of the stash put in the stack
- `stash@{0}` is the identifier of the last stash put in the stack

The remaining text is a description that helps you to infer when the stash was taken and from what branch it comes.

Apply Stashed Contents

We can apply any stash back to the staging area and working area as follows.

First, let's check out the branch we want to operate into – for example, let's check out `f-docs`:

```
git checkout f-docs
```

Since we pushed the stash `stash@{1}` from this branch, we can now apply it back and continue our work as follows:

```
git stash apply stash@{1}
```

The output is as follows:

```
On branch f-docs
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   src/doc/README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    src/doc/Changelog.md

no changes added to commit (use "git add" and/or "git commit -a")
```

As you can see from the output, everything has been restored as it was initially.

Note Mind that applying the stash does not remove it from the list of stashes.

We can now add everything to the staging area and commit it to the repository area:

```
git add --all
git commit -m "Fixed typo in docs"
```

In some scenarios, it may happen that merge conflicts arise while applying a stash: in this case, we can save the contents of stash to a new branch and then merge it as usual (more on this topic later).

For example, the statement:

```
git stash branch f-oddconf stash@{0}
```

creates the new `f-oddconf` branch and extracts the contents of the `stash@{0}` stash into its working area. The statement produces the following output:

```
Switched to a new branch 'f-oddconf'
On branch f-oddconf
```

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   src/etc/oddconf.xml

Dropped stash@{0} (04c219f274c2bf552bdaff919a38830bd4fafb06)
```

As you can infer from the `Dropped stash@{0}` warning, conversely from when applying a stash, this command does also remove the entry from the stack of the stashes.

Indeed, if you run `git stash list` now, you will only get the following entry:

```
stash@{0}: WIP on f-docs: 814a59d Merge branch ...
```

Remove Stashes

Let's clean up the stack of stashes by removing the last entry as follows:

```
git stash drop stash@{0}
```

Mind that if you want to apply the last stash, you can just type `git stash pop`: this command, in addition to applying the stash, also removes it from the stack of stashes.

Let's push the change to the remote origin Gitea repository:

```
git push -u origin f-docs
```

and clean up the workspace:

```
git reset --hard
git checkout main
git branch -d f-oddconf
```

Remote Branches

As we saw while playing with the Gitea web UI, branches can be pushed to remote repositories to share their contents with other developers.

Listing Remote Branches

The most straightforward way to list remote branches is running

```
git branch -r
```

The output is

```
origin/f-docs
origin/f-push-rpm
origin/main
```

In this example, the remote repository, besides the `main` branch, contains also the following `f-push-rpm` and `f-docs` branches.

This must sound odd to you: in the previous chapter, using the Web UI, we raised a pull request for pulling the contents of the `f-push-rpm` branch into the `main` branch, deleting the `f-push-rpm` branch once done, so it should sound strange that branch being still listed.

The reason for this unexpected output is because `git branch -r` lists the remote branches that are already known to Git.

To really connect to the remote repository and retrieve the list of remote branches, we must use `git ls-remote` statement – this statement by default returns all the remote commit hashes, but we can filter it to return only the heads:

```
git ls-remote --heads
```

This time the reported branches are (mind the output is cropped to fit the page width – the end of the last line is `refs/heads/releases/1.0.0`):

```
7a64e8ba35099b363f49638d20aca82020c38904    refs/heads/f-docs
a83965846477e1ef1f69de5b4149db7711e9e060    refs/heads/main
a83965846477e1ef1f69de5b4149db7711e9e060    refs/heads/release
```

As you can see, now the `f-push-rpm` branch is no longer listed.

In addition to the `main` branch, there are also the `f-docs` we just pushed and `release/1.0.0` branches that we created when we tested the Gitea action workflow for publishing a release.

As we have just seen, besides configuring remotes, Git requires us to specify which is the default remote for either pull or push each of the available local branches.

We can display all these settings as a whole by typing `git remote show <remote-label>`. output

```
git remote show origin
```

In this scenario, the output is

```
* remote origin
  Fetch URL: http://192.168.66.177:30001/carcano/fancymonitor.git
  Push  URL: http://192.168.66.177:30001/carcano/fancymonitor.git
  HEAD branch: main
  Remote branches:
      f-docs                     tracked
      main                       tracked
      refs/remotes/origin/f-push-rpm stale (use 'git remote prune'
      to remove)
      release/1.0.0              new (next fetch will store in
                                 remotes/origin)
  Local branches configured for 'git pull':
      f-docs     merges with remote f-docs
      f-push-rpm merges with remote f-push-rpm
      main       merges with remote main
  Local refs configured for 'git push':
      f-docs pushes to f-docs (up to date)
      main   pushes to main   (local out of date)
```

The above output reveals several interesting things:

- The local `main` branch is out of date, so it is necessary to pull changes from the remote origin to make it current.
- The local `f-docs` branch is up to date with the remote origin.
- The local `f-push-rpm` branch is referencing a remote branch that has been deleted, so it is necessary to prune the local settings to have it removed also from local.
- There is a remote `release/1.0.0` branch that at the next fetch process will be added to the references of `remotes/origin`.

Check Out a Remote Branch

It is important to know that, without configuring the remote references, it is not possible to check out a remote branch.

We can easily verify this as follows:

```
git checkout release/1.0.0
```

As you can see, the statement raises the following error:

```
error: pathspec 'release/1.0.0' did not match any file(s) known to git
```

The error is raised because we have not yet fetched the `release/1.0.0` branch from the remote. Let's do it now by running

```
git fetch origin release/1.0.0
```

The important bits from the output are

```
 * branch              release/1.0.0 -> FETCH_HEAD
 * [new branch]        release/1.0.0 -> origin/release/1.0.0
```

As you can see from above, after fetching the remote `origin/release/1.0.0` branch, the branch is then available as `release/1.0.0` local branch.

It is finally possible to check out the branch:

```
git checkout release/1.0.0
```

Note Mind that `git fetch` retrieves remote branches' metadata and the list of commits but does not merge the commits in the working area: if you want to retrieve the remote changes and blend them into the working area you must instead run `git pull`.

Pruning a Branch

As we saw, the output of the `git remote show origin` statement contains:

```
refs/remotes/origin/f-push-rpm stale (use 'git remote prune' to remove)
```

These orphan references can be deleted by running the `git fetch` statement with the `--prune` option as follows:

```
git fetch --prune
```

The output is

```
 - [deleted]     (none)     -> origin/f-push-rpm
```

Please be wary that we just deleted the reference to the non-existent remote, not the local `f-push-rpm` branch: you can easily verify it by running `git branch`.

If you wish to have pruning automatically performed at each fetch operation, just configure the `fetch.prune` setting as follows:

```
git config --global fetch.prune true
```

Pulling Changes from the Upstream Remote Branch

To experiment with pulling changes from the upstream remote branch, let's modify the `src/doc/README.md` file directly in the remote upstream repository.

In the Gitea UI, within the `f-docs` branch, modify the `src/doc/README.md` file – for example, change the last line as follows:

```
# Fancymonitor

This is the official Fancymonitor documentation.
```

When doing online editing of a file, each time you save changes by clicking the `Commmit Changes` button, Gitea creates a new commit.

You can pull new commits of the current checked-out branch from the remote repository. For example, to pull the latest commits of the `f-docs` branch, first check out the `f-docs` branch:

```
git checkout f-docs
```

and then just pull the commits as follows:

```
git pull
```

The output is as follows:

```
From http://192.168.66.177:30001/carcano/fancymonitor
   7a64e8b..4ce3a19  f-docs      -> origin/f-docs
   814a59d..a839658  main        -> origin/main
Updating 7a64e8b..4ce3a19
Fast-forward
```

```
src/doc/README.md | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Note While developing on already existing remote branches, the best practice is to periodically run `pull` statements to have changes pushed from other developers to the remote branch merged into your local copy. Doing this way, you reduce the risk of having you writing conflicting changes, and you also early detect conflicting changes written by other people. Additionally, since the `push` command we are about to explain has been designed to statically push without integrating changes, any changes pushed to that remote branch in the meantime will make your `push` statements fail.

Pushing to Remotes

When necessary (typically right after a commit), it is possible to push local branches to remote upstream repositories.

This operation, besides the obvious intent of sharing the changes with other developers, also has the nice side effect of creating a centralized backup copy of our local changes, protecting us from data loss caused by accidental deletion or disk failure on our own computer.

Push operations are invoked by running the `git push` statement: as anticipated, this is just a static push operation, which does not involve any kind of integration with changes which have been pushed to the remote branch in the meantime.

To see this in action, let's create again the `develop` branch from the `main` branch, checking out to it at the same time:

```
git checkout -b develop main
```

Pushes are carried on by just running `git push`. Anyway, the first time we run a push statement, we must also explicitly tell Git to which is the upstream remote repository for the pushed branch by using the `-u` option. Git will then remember this information, sparing us from having to specify the remote each time.

The basic syntax of the statement to run is `git push -u <remote-alias> <local-branch-name>`.

So, in our example, run

```
git push -u origin develop
```

Of course, if the branch does not exist yet in the remote it gets created.

By now, unless you need to push to a different remote upstream repository and a different remote branch, every next push operation can be simply performed by running

```
git push
```

Note As mentioned, when dealing with already existing remote branches, the best practice is to run a `pull` statement before attempting a `push` – the remote branch indeed may contain conflicting changes making the `push` unmergeable. In such a scenario, you must pull the remote branch, locally resolve the conflicts and the `push`.

The push statement by default assumes you want to push the current branch, but it is of course possible to push multiple branches to the same remote upstream repository within the same statement.

For example, `git push origin f-xml-only f-xml-config` pushes both `f-xml-only` and `f-xml-config` to the `origin` remote upstream repository into their respective remote branches.

If necessary, you can of course push to a remote branch different from the configured default one, even to a different remote upstream repository.

For example, `git push replica f-fancy:f-squashed` push `f-fancy` to the `f-squashed` branch on the `replica` remote upstream repository.

Note Mind that in a clean design it is always best to use the KISS approach, … so both the local and remote branch names must match. I did this extreme example only to show that it is technically possible to handle this use case.

Deleting a Remote Branch

If you need to delete a remote branch, you must use the `git push` statement with the `--delete` command-line option. For example:

```
git push origin --delete f-docs
```

After executing this command, if you run `git ls-remote --heads`, the `f-docs` branch will no longer be listed. The same applies if you check from the Gitea web UI.

Note Since we do not need to operate further as the `build` user, exit the sudoed shell by typing `exit` or pressing `CTRL+D`.

Merging Branches

When working with branches, it quickly becomes common to need to merge the contents of one branch into another.

A typical use case is feature/topic branches: they are usually dedicated to developing a new feature, and when it is done, they are merged back into their parent branch. Another common use case is refreshing a feature/topic branch from the parent, merging the parent's new changes into it.

After the first commit is added to a branch, the branch diverges from its ancestor: the branch's `HEAD` is linked to that commit, and it is moved forward at each further commit. These operations make the branch to be ahead of its ancestor.

When merging you have the following subject and object:

- **Target branch**: The branch you checked out and you are currently in; that is the target of the merge operation indeed.
- **Source branch**: The branch you want to merge the contents from into the target branch. Mind that by default the source branch does not get deleted after the merge, unless you explicitly do it.

Branch Merging Strategies

Git provides several merging strategies, trying to automatically select the more appropriate one depending on if the involved branches diverged or not.

Here we just discuss in detail the most used ones, showing the options for explicitly invoking the desired ones, but mind that there do exist other less used strategies such as the resolve strategy, the octopus strategy and so on.

As usual we illustrate things in action, so let's start by creating and populating another Git repository to have a playground to experiment with.

First, switch to this chapter's workspace directory:

```
cd ~/devsecops-pillars/ch35
```

Then, run the following list of statements:

```
git init git-merging-strategies
cd git-merging-strategies
touch README.md
git add README.md
git commit -m "Initial commit"
echo "# Readme" > README.md
git add README.md
git commit -m "added the heading"
```

To understand the outcome of this baseline, let's have a look at the log:

```
git log --all --decorate --oneline --graph
```

The output is as follows:

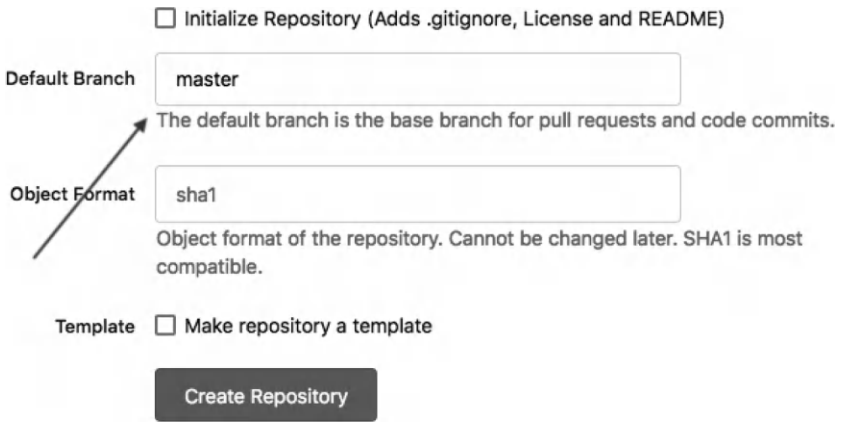
```
* b1bb874 (HEAD -> master) added the heading git-merging-strategies
* 03a67f5 Initial commit
```

Please note how it reports that the current branch is `master`.

Since at the end of the lab we will check the outcome also in the Gitea UI, let's now create the `git-merging-strategies` repository in Gitea within the `Carcano` organization.

It is important to take care of the following:

- Not to tick the checkbox for initializing the repository
- Modify the name of the default branch to match the current one – in this specific case, as we saw it is `master`



Last but not least, as suggested by the `Quick Guide`, configure the justly created Gitea repository as the upstream remote of the local Git repository by running

```
git remote add origin \
http://192.168.66.177:30001/carcano/git-merging-strategies.git
```

As usual, replace the `192.168.66.177` IP address with the one of your Gitea instance, as reported by the `Quick Guide`.

We have now completed the baseline lab and we are now ready to use it for experimenting with the most common branch merging strategies.

Fast-Forward

This is one of the simplest and easiest to understand merging strategies – it can be performed only when

- Both branches have a common commit, that is, an ancestor of the source branch
- The ancestor is the last commit of the target branch

In such a scenario, the commits are just added to the target branch, and the target branch's head is moved to the last commit – hence the name fast-forward.

Unfortunately, unless you are working on local branches, this is quite a rare use case.

Performing a fast-forward merge just requires you to be already in the target branch, and then run the `git merge` command with the `--ff-only` option.

For example:

```
git merge --ff-only source-branch
```

Let's see this in action on the `git-merging-strategies` Git repo we created as a playground.

Create the `f-fast-forward-example` branch, then modify the contents a few times committing each single change:

```
git checkout -b f-fast-forward-example
echo "1st change" > fast-forward.txt
git add fast-forward.txt; git commit -m "fast-forward - 1st commit"
echo "2nd change" >> fast-forward.txt
git add fast-forward.txt; git commit -m "fast-forward - 2nd commit"
echo "3rd change" >> fast-forward.txt
git add fast-forward.txt; git commit -m "fast-forward - 3rd commit"
```

Let's check the commit history by plotting an ASCII graph:

```
git log --all --decorate --oneline --graph
```

The output is as follows:

```
* 27e3d5e (HEAD -> f-fast-forward-example) fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 (master) added the heading
* 03a67f5 Initial commit
```

As you can see, the log reports both branches, and they are linked to a different commit id – `master` is bound to `b1bb874`, whereas `f-fast-forward-example` is bound to `27e3d5e`.

Let's now run the fast-forward merge – first we check out the `master` branch:

```
git checkout master
```

Then we run the actual merge:

```
git merge --ff-only f-fast-forward-example
```

You should see a message like the below one, indicating that a fast-forward occurred:

```
Updating b1bb874..27e3d5e
Fast-forward
 fast-forward.txt | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 fast-forward.txt
```

Note In scenarios where a fast-forward merge is possible, Git will automatically select the fast-forward strategy. If you want to prevent a fast-forward merge – such as when implementing the GitFlow branching model – you must explicitly tell Git not to use the fast-forward strategy by passing the `--no-ff` option. For example: `git merge --no-ff f-source-branch`.

Let's now check how the log changed after the merge:

```
git log --all --decorate --oneline --graph
```

this time the output is as follows:

```
* 27e3d5e (HEAD -> master, f-fast-forward-example) fast-forward -
3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

As you can see, now both `master` and `f-fast-forward-example` point to the `27e3d5e` commit.

Let's now delete the branch, to have a clean workplace for the next merge strategy example.

```
git branch -d f-fast-forward-example
```

Rebase and Fast-Forward

As mentioned, one of the two mandatory conditions for being able to do a fast-forward merge is that the ancestor commit of the branches must also be the last commit of the target branch.

This condition is rarely met in practice, especially in collaborative environments where multiple commits are made to different branches.

However, if both branches have a common ancestor, you can use the rebase feature to relink the source branch to the tip of the target branch, allowing for a fast-forward merge.

This is a two-step process – first, check out the source branch and rebase it onto the target branch:

```
git checkout source-branch
git rebase target-branch
```

After rebasing, switch to the target branch and perform the fast-forward merge:

```
git checkout target-branch
git merge --ff-only source-branch
```

Let's see in action using the `git-merging-strategies` Git repo we created as a playground.

We start from having an initial look at the log:

```
git checkout master
git log --all --decorate --oneline --graph
```

This time the output is as follows:

```
* 27e3d5e (HEAD -> master) fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

In this example, we create the `f-rebase-and-ff` branch and modify the contents a few times committing at each single change:

```
git checkout -b f-rebase-and-ff
echo "1st change" > f-rebase-and-ff.txt
git add f-rebase-and-ff.txt; git commit -m "f-rebase-and-ff - 1st commit"
echo "2nd change" >> f-rebase-and-ff.txt
git add f-rebase-and-ff.txt; git commit -m "f-rebase-and-ff - 2nd commit"
```

then, we check out the `master` branch again, modifying the contents a few times committing at each single change:

```
git checkout master
echo "4th change" > fast-forward.txt
git add fast-forward.txt; git commit -m "master - 1st commit"
echo "5th change" >> fast-forward.txt
git add fast-forward.txt; git commit -m "master - 2nd commit"
```

Lastly, let's check out the `f-rebase-and-ff` branch again, modify the contents and commit the change:

```
git checkout f-rebase-and-ff
echo "3rd change" > f-rebase-and-ff.txt
git add f-rebase-and-ff.txt; git commit -m "f-rebase-and-ff - 3rd commit"
```

We made enough changes to provide a good scenario for illustrating how to operate this merge.

Let's check how the history looks like now:

```
git log --all --decorate --oneline --graph
```

this time the output is as follows:

```
* e64761e (HEAD -> f-rebase-and-ff) f-rebase-and-ff - 3rd commit
* 3ed8e82 f-rebase-and-ff - 2nd commit
* f78be98 f-rebase-and-ff - 1st commit
| * 46bbd7a (master) master - 2nd commit
| * ee9400f master - 1st commit
|/
```

```
* 27e3d5e fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

As you can see, we have two branches, `master` and `f-rebase-and-ff`, both originating from the `27e3d5e` commit.

By following the graph's lines, we can infer that

- `ee9400f` and `46bbd7a` commits belong to the `master` branch
- `f78be98`, `3ed8e82` and `e64761e` commits belong to the `f-rebase-and-ff` branch
- Every commit before `27e3d5e`, along with the `27e3d5e` commit itself, belongs to both branches

Indeed, if we check the difference in the number of commits from the `27e3d5e` and the HEAD of the `master` branch

```
git rev-list --left-right --count 27e3d5e...master
```

we get

```
0    2
```

This means that `master` is 2 commits ahead of the `27e3d5e` commit.

We can do the same for the `f-rebase-and-ff` branch:

```
git rev-list --left-right --count 27e3d5e...f-rebase-and-ff
```

This time the output is

```
0    3
```

So `f-rebase-and-ff` is 3 commits ahead of the `27e3d5e` commit.

Since both branches diverged from their ancestor (`27e3d5e`), it is not possible to run a fast-forward merge.

Indeed, if we check out the master branch

```
git checkout master
```

and attempt a fast-forward merge

```
git merge --ff-only f-rebase-and-ff
```

The outcome is just raising the following error:

```
fatal: Not possible to fast-forward, aborting.
```

Let's now try the rebase and fast-forward merging strategy.

First, rebase the `f-rebase-and-ff`, moving its ancestor to the `master`'s branch current tip, running

```
git checkout f-rebase-and-ff
git rebase master
```

Note This is the point where maybe conflicts of content version arise: if Git cannot automatically solve them, you must manually reconcile them.

Let's check how this operation affected the history:

```
git log --all --decorate --oneline --graph
```

This time the output is as follows:

```
* c7c51dd (HEAD -> f-rebase-and-ff) f-rebase-and-ff - 3rd commit
* 8577c44 f-rebase-and-ff - 2nd commit
* d624b97 f-rebase-and-ff - 1st commit
* 46bbd7a (master) master - 2nd commit
* ee9400f master - 1st commit
* 27e3d5e fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

As you can see we still have two branches, `master` and `f-rebase-and-ff`, but now `f-rebase-and-ff` originates from the `46bbd7a` commit.

Since now it is possible, we can re-try the fast-forward merge:

```
git checkout master
git merge --ff-only f-rebase-and-ff
```

This time it works like a charm.

Let's have a look at how the log changed:

```
git log --all --decorate --oneline --graph
```

This time the output is as follows:

```
* c7c51dd (HEAD -> master, f-rebase-and-ff) f-rebase-and-ff - 3rd commit
* 8577c44 f-rebase-and-ff - 2nd commit
* d624b97 f-rebase-and-ff - 1st commit
* 46bbd7a master - 2nd commit
* ee9400f master - 1st commit
* 27e3d5e fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

Let's now delete the branch, to have a clean workplace for the next merge strategy example.

```
git branch -d f-rebase-and-ff
```

Ostensibly Recursive Twin (ORT) Merge

The downside of fast-forward merging strategies is that they create a linear history, which can obscure the actual track of commits. If your goal is to maintain a clear record of the commit history, you should use a recursive merge strategy, such as the new `ORT` strategy or the older `recursive` strategy.

Let's see in action using the `git-merging-strategies` Git repo we created as a playground.

Let's check out the `f-recursive` feature from the `master` branch, modify the contents a few times and commit at each single change:

```
git checkout -b f-recursive
echo "1st change" > recursive.txt
git add recursive.txt; git commit -m "recursive - 1st commit"
echo "2nd change" >> recursive.txt
git add recursive.txt; git commit -m "recursive - 2nd commit"
```

Then check out the `master` branch, modify the contents a few times, and commit at each single change:

```
git checkout master
echo "6th change" > fast-forward.txt
git add fast-forward.txt; git commit -m "master - 3rd commit"
echo "7th change" >> fast-forward.txt
git add fast-forward.txt; git commit -m "master - 4th commit"
```

Lastly, let's check out the `f-recursive` branch again, modify the contents, and commit them:

```
git checkout f-recursive
echo "3rd change" > recursive.txt
git add recursive.txt; git commit -m "recursive - 3rd commit"
```

and have a look at the history:

```
git log --all --decorate --oneline --graph
```

The output should look as follows:

```
* 57a776e (HEAD -> f-recursive) recursive - 3rd commit
* 639786e recursive - 2nd commit
* a087fea recursive - 1st commit
| * 557ca7f (master) master - 4th commit
| * 473c86a master - 3rd commit
|/
* c7c51dd f-replace-and-ff - 3rd commit
* 8577c44 f-replace-and-ff - 2nd commit
* d624b97 f-replace-and-ff - 1st commit
* 46bbd7a master - 2nd commit
* ee9400f master - 1st commit
```

```
* 27e3d5e fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

That is exactly the same scenario we had in the rebase and fast-forward merge: as you can see we have two branches, `master` and `f-recursive`, both originating from the `c7c51dd` commit.

By following the graph's lines, we can infer that

- `473c86a` and `557ca7f` commits belong to the `master` branch
- `a087fea`, `639786e`, and `57a776e` commits belong to the `f-recursive` branch
- Every commit before `c7c51dd`, along with the `c7c51dd` commit itself, belongs to both branches.

If we check the difference in the number of commits from the `c7c51dd` and the HEAD of the `master` branch

```
git rev-list --left-right --count c7c51dd...master
```

we get

```
0    2
```

This means that `master` is 2 commits ahead of the `c7c51dd` commit.

In such a scenario, since the fast-forward strategy is not viable, if you don't explicitly specify the merging strategy, Git automatically proceeds with the `ort` (or `recursive`) merge.

Let's now attempt the merge as follows

```
git checkout master
git merge -s ort f-recursive
```

In this example, I'm intentionally adding the `-s ort` to show the explicit invocation of the `ort` strategy so that you know how to do that as needed but, as stated above, this strategy is the one attempted by default by Git in such a scenario, so here it is not really necessary to specify `-s ort`.

Note The `ort` strategy has been introduced as the default after git version 2.34 as an improvement of the former `recursive` strategy – if you are dealing with an older Git version, you must specify `-s recursive` instead.

Since this causes a new commit to be created, a text editor with the commit text is opened: just write the commit message and save it – if it is `vi`, just switch to command mode pressing the ESC key, then type `:x`.

The final output is

```
Merge made by the 'ort' strategy.
 recursive.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 recursive.txt
```

As you can see, the outcome of this merge is a new commit with the merged contents of the two branches.

Note If there are any conflicts during the merge, Git will prompt you to resolve them manually. After resolving conflicts, you will need to commit the merge.

```
git log --all --decorate --oneline --graph
```

This time the output is as follows:

```
*   a19da7c (HEAD -> master) Merge branch 'f-recursive'
|\
| * 57a776e (f-recursive) recursive - 3rd commit
| * 639786e recursive - 2nd commit
| * a087fea recursive - 1st commit
* | 557ca7f master - 4th commit
* | 473c86a master - 3rd commit
|/
* c7c51dd f-replace-and-ff - 3rd commit
* 8577c44 f-replace-and-ff - 2nd commit
* d624b97 f-replace-and-ff - 1st commit
```

```
* 46bbd7a master - 2nd commit
* ee9400f master - 1st commit
* 27e3d5e fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

Now, let's delete the `f-recursive` branch:

```
git branch -d f-recursive
```

If we check the log again now:

```
git log --all --decorate --oneline --graph
```

This time the output is as follows:

```
*   a19da7c (HEAD -> master) Merge branch 'f-recursive'
|\
| * 57a776e recursive - 3rd commit
| * 639786e recursive - 2nd commit
| * a087fea recursive - 1st commit
* | 557ca7f master - 4th commit
* | 473c86a master - 3rd commit
|/
* c7c51dd f-replace-and-ff - 3rd commit
* 8577c44 f-replace-and-ff - 2nd commit
* d624b97 f-replace-and-ff - 1st commit
* 46bbd7a master - 2nd commit
* ee9400f master - 1st commit
* 27e3d5e fast-forward - 3rd commit
* e47e3ac fast-forward - 2nd commit
* f9b719e fast-forward - 1st commit
* b1bb874 added the heading
* 03a67f5 Initial commit
```

Although the `f-recursive` branch has now disappeared since it was deleted, the graph and the commit message let us still infer it existed, and that its commits were merged into the `master` branch in the `a19da7c` commit.

Squashed Merge

This is the last of the merging strategies described in this book: it combines all the commits from the source branch into a single commit in the target branch.

This approach is useful for maintaining a clean and concise commit history, as it "squashes" multiple commits into one.

Please note that it stops right after merging without creating the actual commit, so enable checking the contents and adding further changes as necessary.

Let's see in action using the `git-merging-strategies` Git repo we created as a playground.

Let's start from checkout the new `f-squashed` branch from the `master` branch, modify the contents a few times and commit at each single change:

```
git checkout -b f-squashed
echo "1st change" > squashed.txt
git add squashed.txt; git commit -m "squashed - 1st commit"
echo "2nd change" >> squashed.txt
git add squashed.txt; git commit -m "squashed - 2nd commit"
```

Then check out the `master` feature again, modify the contents, and commit:

```
git checkout master
echo "8th change" > fast-forward.txt
git add fast-forward.txt; git commit -m "master - 5th commit"
```

Lastly, let's check out the `f-squashed` branch again, modify the contents, and commit:

```
git checkout f-squashed
echo "3rd change" >> squashed.txt
git add squashed.txt; git commit -m "squashed - 3rd commit"
```

To figure out the current status, let's have a look at the log again – this time, to limit the output, we set the boundaries focusing only on the commits between the `master` branch and the `f-squashed` branch:

```
git log --decorate --oneline --graph --boundary master...f-squashed
```

The output is as follows

```
* 6fea6d8 (HEAD -> f-squashed) squashed - 3rd commit
* d2eb3a1 squashed - 2nd commit
* 95fb0bf squashed - 1st commit
| * c84bc77 (master) master - 5th commit
|/
o a19da7c Merge branch 'f-recursive'
```

As you can see, `a19da7c`, that is the ancestor of the `f-squashed` branch, is behind the `master`, indeed:

```
git rev-list --left-right --count a19da7c...master
```

returns the following output:

```
0    1
```

Let's check the files currently extracted from the repository:

```
ls -1
```

The output is as follows:

```
fast-forward.txt
f-rebase-and-ff.txt
README.md
recursive.txt
squashed.txt
```

We can now squash all the commits currently in the `f-squashed` branch into the `master` branch by running

```
git checkout master
git merge --squash f-squashed
```

The output contains the following messages:

```
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Note This is the point where maybe conflicts of content version arise, and if Git cannot automatically solve them you must manually reconcile them.

Let's have a look at the contents of the staging area:

```
git status
```

The output is as follows:

```
On branch master
Changes to be committed:
  (use "git restore --staged ..." to unstage)
     new file:   squashed.txt
```

So, as expected, the `squashed.txt` file, coming from the `f-squashed` branch, has been added to the staging area.

As you can easily argue, it has been added to the working area too:

```
ls -1
```

The output is as follows indeed:

```
fast-forward.txt
f-rebase-and-ff.txt
README.md
recursive.txt
squashed.txt
```

As for the contents of the `squashed.txt` file

```
cat squashed.txt
```

the output is

```
1st change
2nd change
3rd change
```

So, to summarize, all of the three commits have been merged into the `master` branch, … but the changes are still uncommitted.

This enables us to do further processing as necessary. When we are satisfied, we can proceed with the commit:

```
git commit -m "squashed f-squashed branch into master branch"
```

This time the output is

```
[master 7244c80] squashed f-squashed branch into master branch
 1 file changed, 3 insertions(+)
 create mode 100644 squashed.txt
```

As we did so far, let's delete the `f-squashed` branch:

```
git branch -d f-squashed
```

This time the operation fails with the following error:

```
error: The branch 'f-squashed' is not fully merged.
If you are sure you want to delete it, run 'git branch -D f-squashed'.
```

This is expected: since we squashed all the commits as a single change into the `master` branch, Git points out that there are commits that have never been merged and so asks if we are sure about what we are doing.

Since in this case it is expected, we can force the deletion of the branch using the capital "d" as suggested by the message:

```
git branch -D f-squashed
```

Revert a Merge

Same way as you may need to revert a commit, it may be necessary to revert a merge, so to undo the changes introduced by that merge.

The approach depends on whether the target branch has already been pushed to a remote repository:

- If the target branch has not been already pushed, you can use `git reset --hard <commit>` to restore commits on the target branch before the merge. Be careful with this command, as it will erase all changes made after the specified commit, including the commit history. Be wary that this action wipes the working area and rewrites the Git history, so make sure not to have any uncommitted changes or important commits that you want to keep.

- If the target branch has already been pushed, you must instead rely on `git revert`. After locating the commit id of the merge you want to revert by running `git log --oneline`, assuming that `merge_commit_id` is the commit id of the merge you want to revert, you can run either
  - `git revert -m 1 <merge_commit_id>` to bring the branch back to its pre-merge state on the target branch or
  - `git revert -m 2 <merge_commit_id>` to reverse the merge in a way that might incorporate or undo target-branch changes – mind that this strategy is rarely used, as it can lead to confusing results

An alternate approach is to treat it like reverting a commit running `git log` without options: this way it also shows the merge-related commits, enabling you to directly use them in the `git revert`. The downside of this method is that, since you are not actually reverting the merge (you are instead reverting the commit), you will have to manually resolve conflicts.

Each merge commit will have a `Merge` line listing the short commit IDs of the target and source branches. For example:

```
commit a19da7cf6d3e1352af949c21650d0e2e18b1d38b
Merge: 557ca7f 57a776e
Author: Marco Antonio Carcano <marco.carcano@carcano.corp>
Date:   Sun May 25 09:17:25 2025 +0000

    Merge branch 'f-recursive'
```

In this case, you can simply run `git revert` followed by the short commit id of the branch you want to revert.

Note Reverting fast-forward merges can be cumbersome because there is no merge commit to revert. If the target branch has already been pushed, you cannot use `git reset --hard` safely. Instead, you would need to revert each individual commit in reverse order to restore the branch to its previous state.

Let's use the above Git log snippet as an example to see how to revert a merge by leveraging on reverting the commit and manually resolving the conflicts: from the above output we can guess that the `f-recursive` branch (`57a776e`) has been merged into the `master` (`557ca7f`) branch.

Let's try reverting:

```
git revert 557ca7f
```

Looks like we are out of luck:

```
error: could not revert 557ca7f... master - 4th commit
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git revert --continue".
hint: You can instead skip this commit with "git revert --skip".
hint: To abort and get back to the state before "git revert",
hint: run "git revert --abort".
```

As expected, the conflict is raised.

Resolve Conflicts

To resolve the conflict, we must first check the current status, to be able to figure out which files are affected by the conflict.

Just run

```
git status
```

The interesting bit of the output is

```
Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
    both modified:   fast-forward.txt
```

which informs us that the `fast-forward.txt` file was modified in both commits.

Let's have a look at its contents.

```
cat fast-forward.txt
```

The output is

```
<<<<<<< HEAD
8th change
=======
6th change
>>>>>>> parent of 557ca7f (master - 4th commit)
```

Since we are reverting to the parent of `557ca7f` commit, we want to keep the line containing the `6th change`.

Edit the file until its content is just:

```
6th change
```

Save the change, then add the file to the staging area and continue the revert process by running

```
git add fast-forward.txt
git revert --continue
```

At the completion of the revert process, a new commit is automatically created: just write the commit message and save it.

The final output is

```
[master e393c80] Revert "master - 4th commit"
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Note The same process we did for resolving the conflict happened during the revert operation can be applied to resolve merge conflicts – the only difference is the last statement, that when dealing with resolving a merge conflict is `git merge --continue`.

If you have a look at the Git history, you will find a new commit generated by this revert.

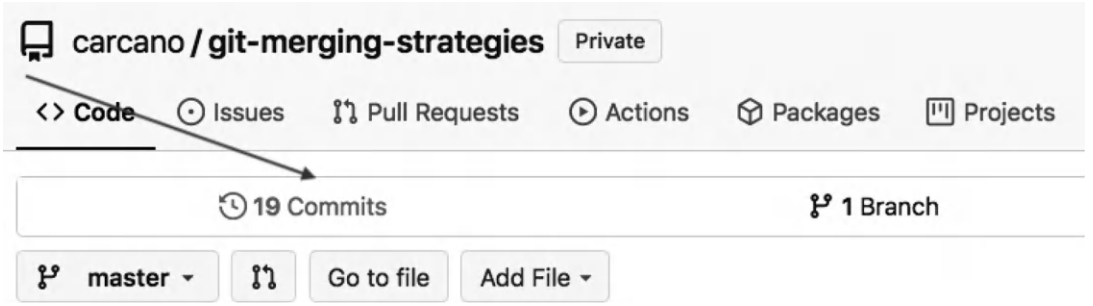
This time, instead of running the `git log` statement, let's have a look at the Git history using Gitea.

First, push the changes so to have them available in Gitea:

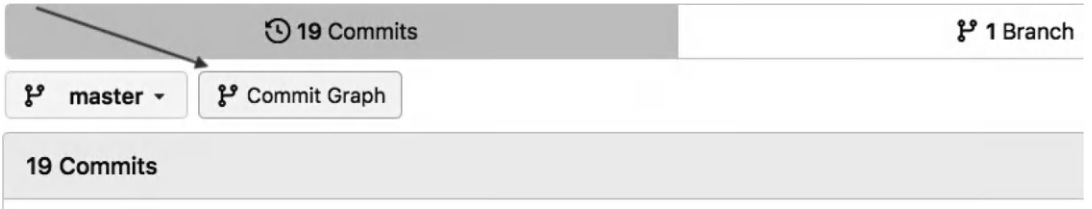
```
git push -u origin master
```

Then, using the Gitea web UI, navigate to the `Carcano` organization and open the `git-merging-strategies` repository.

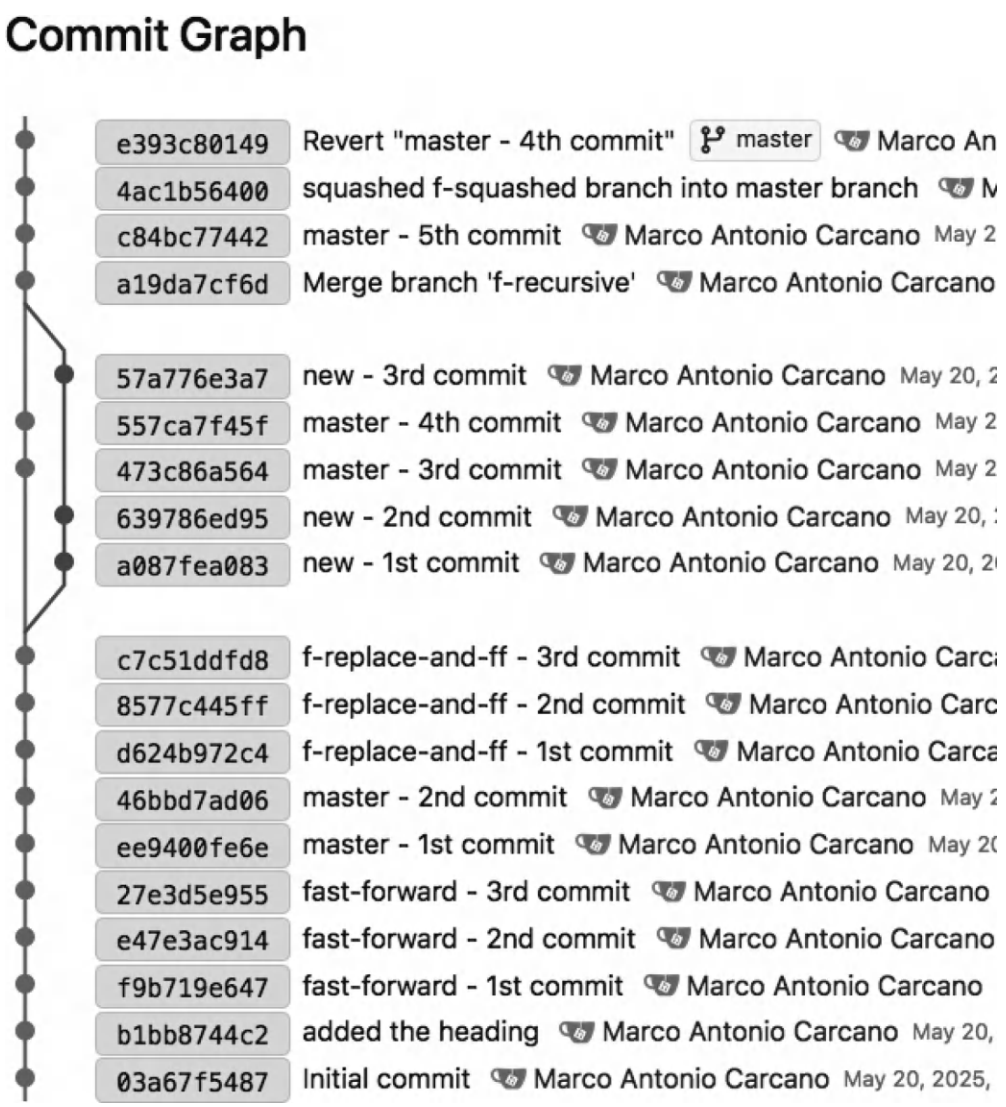
Once the main page of the `git-merging-strategies` repository is displayed, click on the link with the number of commits as shown by the below picture:



This opens the page with the list of commits: click on the `Commit Graph` as shown by the below picture:



This finally opens the page showing the commit graph, as shown by the below picture:



Tagging

As we already mentioned, tags are names used to conveniently identify one and only one hash: their purpose is to provide a human-readable name to refer to a specific hash so as to ease checking out contents of a given point of the story.

This means that tags have a repository-wide scope (they do not belong to any branch), and so their names must be unique across the repository.

Note In addition to clearly identifying software releases, tags are valuable in configuration management. They allow you to refer to a specific state of the repository's contents whenever needed, without worrying about changes made to the repository in the meantime.

To better illustrate their usage, let's have a quick walkthrough to see them in action – switch to the `build` user:

```
sudo su - build
```

and change to the Fancymonitor project root directory:

```
cd /opt/build/fancymonitor
```

Creating a Tag

To tag the current commit simply type `git tag` followed by the name you want to assign to the tag and an optional descriptive message.

As an example, let's create the `1.0.0` tag with the message `Release 1.0.0`.

First, since we already have the `release/1.0.0` branch dedicated to release `1.0.0`, we must check out it as follows:

```
git checkout release/1.0.0
```

then we can tag the current commit id by running

```
git tag 1.0.0x -m "Release 1.0.0"
```

Listing Tags

You can list the available tags by typing

```
git tag
```

The output shows we clearly made a typo:

```
1.0.0x
```

Deleting a Tag

If you want to delete a tag, type `git tag -d` followed by the name you want to delete.

For example, we can delete the typo made when creating the `1.0.0` tag:

```
git tag -d 1.0.0x
```

and re-create it with the correct name:

```
git tag 1.0.0 -m "Release 1.0.0"
```

Note Mind that when deleting the tag, you are not removing the hash of the commit it is pointing to, so the contents of the commit are still in the specific branch of the Git repository.

Checking Out a Tag

A tagged commit can be checked out simply by running `git checkout tags/<tag>`.

For example, to check out the tag `1.0.0`, type

```
git checkout tags/1.0.0
```

Remote Tags

Same way as branches, also tags can be pushed to a remote repository.

Pushing Tags to a Remote Repository

As an example, let's push it the tag `1.0.0` to the remote `origin`:

```
git push origin tags/1.0.0
```

the output is as follows:

```
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 170 bytes | 85.00 KiB/s, done.
```

```
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
remote: . Processing 1 references
remote: Processed 1 references in total
To http://192.168.66.177:30001/carcano/fancymonitor.git
 * [new tag]          1.0.0 -> 1.0.0
```

Let's then suppose we made some changes, and we want to tag the release `1.0.1`:

```
git tag 1.0.1 -m "Release 1.0.1"
```

then, let's push it to the remote `origin`:

```
git push origin tags/1.0.1
```

The output is as follows:

```
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
remote: . Processing 1 references
remote: Processed 1 references in total
To http://192.168.66.177:30001/carcano/fancymonitor.git
 * [new tag]          1.0.1 -> 1.0.1
```

Note As you can see, to push a specific tag, you must always prepend `tags/` to the name of the tag, otherwise Git infers that the name of the object you are pushing belongs to a branch.

If necessary, you can also push all the tags that have been added as a whole. To try this, let's first create some other tags:

```
git tag 1.0.2
git tag 1.0.3
```

and then let's push them as a whole:

```
git push --tags
```

Listing Remote Tags

Remote tags can be listed as follows:

```
git ls-remote --tags
```

Deleting Tags from the Remote Repository

Sometimes it is necessary to delete a tag from a remote repository: you can easily achieve this by using the `--delete` option of the `git push` command.

For example, to delete the `1.0.3` tag from the `origin` remote repository, type

```
git push --delete origin tags/1.0.3
```

The output is

```
To http://192.168.66.177:30001/carcano/fancymonitor.git
 - [deleted]         1.0.3
```

Hooks

As anticipated in Chapter 18, Git provides a set of hooks that can be exploited to intercept events and perform additional actions.

You can find sample files beneath the `.git/hooks` directory.

A really handy one is the `pre-commit` hook that is triggered when running commit statements, before performing the actual commit: this enables running custom checks and aborting the commit as necessary.

As you certainly inferred, the `pre-commit` hook is the one exploited by the `pre-commit` Python module we installed in Chapter 18.

As discussed, once the Python module is installed, it is configured on a per-repository basis by placing the `.pre-commit-config.yaml` settings file in the Git's repository root directory and running `pre-commit install`.

Note Git hooks are not managed by `git` – this does not only mean they are not versioned, but also that they are not synchronized to remotes. If you want to exploit hooks on remote servers, you must put the triggered scripts directly on the remote server. Since they can be triggered on push, they must be designed to unroll commits since the last push and perform the checks on each single unrolled commit one by one.

While it is possible to exploit server-side hooks for triggering custom scripts, writing these scripts is much more challenging: for example, processing an `on-push` event to perform a check on the pushed commits requires at least to unroll the pushed commits back to the current branch's tip using the `git rev-list <cur_commit_id> <latest_commit_id>` statement, where

- `<cur_commit_id>` is the commit ID of the tip of the branch on the server before the push. It represents the last commit that was present in the branch before any new commits were pushed.
- `<latest_commit_id>` is the commit ID of the tip of the branch after the push. It represents the latest commit that has been pushed to the branch.

This enables checking each single commit message for running compliance checks.

If instead it is needed to run checks on the actual changes of each commit, the process is even more complex, since it is also necessary to extract a copy of the repo in a temporary workspace. To implement it, it is possible to use `git archive` to check out only the changed files from each single commit and then perform the necessary check, such as running a linter.

Summary

This chapter provides you with a thorough overview of what Git is and how to operate it in nearly every scenario you are likely to encounter.

We explored both theoretical concepts and hands-on practices to create and manage Git repositories, including working with local clones and synchronizing content with remotes as necessary.

Additionally, we delved into the most used merging strategies and learned how to effectively undo changes, whether they involve single commits or merges.

These skills will let you effectively collaborate with cross-functional teams, manage code changes, and ensure the integrity of the software development lifecycle, which is one of the core skills expected by every DevOps and DevSecOps professional.

PART XIV

Remote Configuration Management

Within scalable enterprise infrastructures, managing configurations through local, manual access introduces scale inefficiencies and untraceable configuration drift. Scaling distributed architectures mandates the capacity to bring thousands of nodes into a predictable state while eliminating human error. To achieve this, professionals must reject manual, imperative execution antipatterns—the operational equivalent of blind button-pressing—and treat transport layers and configuration engines deterministically. This section details how to leverage OpenSSH as a secure, automated transport layer and deploy Ansible driven by native idempotence to achieve verifiable transitions of state across environments. By standardizing playbook architectures, isolating infrastructure credentials utilizing Ansible Vault, and integrating public assets from the online Ansible Galaxy registry, you will establish the technical framework required to orchestrate entire data centers within a secure, audit-ready Infrastructure as Code (IaC) framework.

CHAPTER 36

An Overview of SSH

The SSH protocol is not just a means of connecting to a remote host to execute shell commands or securely transfer files; it also serves as a transport protocol for various applications, including Git and Ansible.

This chapter aims to provide an overview of SSH from the client's perspective, focusing on effective usage. We will cover essential concepts such as how the client verifies the identities of SSH servers, the mechanics of public key authentication, the process of generating and authorizing user SSH keys, and how to utilize the SSH agent. Additionally, we will discuss the configuration of the SSH client configuration file.

Furthermore, we will explore the most used security-related directives in OpenSSH server configuration, enabling you to learn how to minimize the attack surface.

This chapter provides a brief, blended summary of concepts drawn from the Secure Shell (SSH) protocol and related tools. It is intended solely as an introductory overview to explain the existence and basic application of SSH in a DevOps and DevSecOps context. This is not a comprehensive or official guide, and it does not replace the original sources. For a deeper understanding, I strongly recommend consulting the primary materials directly:

- **For SSH protocol and OpenSSH details**: Official OpenSSH documentation at openssh.com or *SSH Mastery: OpenSSH, PuTTY, Tunnels, and Keys* by Michael W. Lucas
- **For SSH security and best practices**: SSH.com Academy resources

Readers are encouraged to explore these works for accurate, detailed information.

This chapter includes several labs. To minimize distractions from repeatedly setting up the operating environment before each example, we will establish a baseline lab configuration now.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_36

Some practical examples will require SSH connections to the lab virtual machine (VM) using its fully qualified domain name (FQDN). Since our setup does not include DNS to support name resolution, we need to ensure the FQDN is properly configured in the `/etc/hosts` file. If it is missing, we must add it.

Although the official method to detect the hostname's FQDN is by running the command `hostname -f`, because of the specific design of our labs, it is instead mandatory to obtain it from the `HOSTNAME` environment variable by running

```
echo ${HOSTNAME}
```

In my case, the output is

```
lab01.carcano.corp
```

Next, check the `/etc/hosts` file and look for the above FQDN. It should appear alongside the existing entry for `reg.t1.carcano.corp`. For example, in my setup, the relevant line looks like this:

```
192.168.66.177 reg.t1.carcano.corp lab01.carcano.corp
```

Note If your `/etc/hosts` file already contains your VM's FQDN on any other line – including those starting with the loopback IP addresses (such as `127.0.0.1` or `::1`) – please remove it from those lines. Modify the file so that the FQDN appears only on the line described above.

SSH Basics

The Secure Shell (SSH) was designed decades ago as a secure replacement for Telnet and the BSD "r-commands": these applications were completely lacking confidentiality features, since they were transmitting data, including passwords, in plaintext.

The term "shell" in SSH can be misleading: while SSH can provide access to a command-line shell (like BASH or C shell) on a remote machine, it is primarily a protocol suite that includes various applications for secure communication over a network.

SSH addresses the following security issues:

- **Eavesdropping**: The entire session is encrypted, preventing the interception of passwords and data.
- **Session hijacking**: An attacker cannot take over an existing connection due to the cryptographic techniques used to perform integrity checks.

By default, the SSH protocol uses the registered port TCP 22 and the specifications distinguish between two major versions:

- SSH1
- SSH2

SSH2 is the more secure and widely used version. It introduces features such as improved encryption algorithms and support for public key authentication.

In Linux, the SSH server is managed by a daemon called `sshd`, which listens for incoming connections on the designated port.

SSH security model leverages public/private key pairs.

For example, it uses asymmetric keys for the following:

- Verifying host identity to ensure the client is connecting to the correct server
- Establishing an encrypted communication channel
- As an optional authentication method to validate the connecting user

Host Keys

SSHd host keys are an integral part of the cryptographic technologies used by the protocol. Their usage is indeed twofold:

- Enable the client to securely identify the server by checking the fingerprint of the host key, or, if working in `rhost` mode, enable the server to identify the client host.
- Securely exchange the symmetric key that is used to encrypt the traffic during the session.

An SSH server always has more than just one host key, to be able to provide the client a compatible one with the highest degree of security support by both client and server.

In Red Hat-based systems, the host key files are stored beneath the `/etc/ssh` directory.

Let's have a closer look at them:

```
sudo ls -1 /etc/ssh/*key*
```

The output is:

```
/etc/ssh/ssh_host_ecdsa_key
/etc/ssh/ssh_host_ecdsa_key.pub
/etc/ssh/ssh_host_ed25519_key
/etc/ssh/ssh_host_ed25519_key.pub
/etc/ssh/ssh_host_rsa_key
/etc/ssh/ssh_host_rsa_key.pub
```

Host Key Fingerprint

At connection setup time, SSH clients check the host key fingerprints to verify the identity of the SSH server.

The client performs a lookup of the server's FQDN or IP address used in the connection statement.

First, it seeks in local files in the following order:

- `~/.ssh/known_hosts` for user-defined known hosts
- `/etc/ssh/ssh_known_hosts` for system-wide known hosts

If nothing matches and the `VerifyHostKeyDNS` client option is set to `yes`, then the client performs a DNS lookup for the `SSHFP` records of the servers' FQDN or IP address.

If any of the above lookups return any fingerprints, they are checked with the ones provided by the server, immediately terminating the connection if there is a mismatch.

If instead the lookup returns nothing, then the SSH client prompts asking if you want to trust the fingerprint provided by the server.

For example, let's attempt for the first time a connection to `lab01.carcano.corp` by running

```
ssh lab01.carcano.corp
```

We get prompted with the following question:

```
The authenticity of host 'lab01.carcano.corp (192.168.66.177)' can't be
established.
ECDSA key fingerprint is MD5:1c:05:d9:c4:be:54:ed:4e:d3:42:4f:51:2
b:4f:a9:22.
Are you sure you want to continue connecting (yes/no)?
```

If we type `yes`, then the fingerprint is considered trusted, and so an entry with the FQDN (or IP) along with the fingerprint is added to the `~/.ssh/known_hosts` text file.

From this point on, subsequent connections to that server won't bother you about verifying the identity of this server again.

To complete this lab, let's type `yes` and then type `CTRL+C` to terminate the connection.

Note As you certainly inferred, the first connection is the most critical moment, since you are deciding whether to mark a fingerprint as trusted: at this moment you are vulnerable to a "man in the middle" attack. For this reason, when providing access information to users, besides the usual information such as server FQDN, security best practices require providing also the list of fingerprints so that the connecting user can know if the fingerprint of the server is the expected one.

Getting the Host Key Fingerprint

If you are the administrator running the SSH server, you are supposed to provide the host key fingerprints, either by publishing them in the `SSHFP` DNS records of the servers' FQDN or IP address or just providing them along with the other connection details given to the users.

It is possible to display the host key fingerprints by typing

```
find /etc/ssh -name "*key*".pub -exec ssh-keygen -lf {} \;
```

On my system the output is as follows:

```
2048 b6:b8:0e:0f:14:cb:1c:09:49:b6:ef:0d:3b:85:20:cd   (RSA)
256 1c:05:d9:c4:be:54:ed:4e:d3:42:4f:51:2b:4f:a9:22   (ECDSA)
256 b2:0e:f7:8e:8f:27:e8:c2:b5:db:78:90:de:2d:b0:d8   (ED25519)
```

Authentication Mechanisms

The two most common user authentication mechanisms used by SSH are password-based authentication and public key authentication.

Password-Based Authentication

It is the classic username/password pair: when connecting, the user just gets a password prompt. This is the default authentication method.

Although there also exist facilities, such as `ssh-pass`, to inject the password at authentication time, be wary they add weakness to the overall SSH security model, so they must be used only with extreme caution and only when it is absolutely necessary.

Public Key Authentication

When public key authentication is enabled, the authentication process first attempts to verify the user's identity by using the supplied public key.

If the public key authentication succeeds, the server grants access without checking the user's password.

This authentication mechanism provides some advantages, such as

- It enables authorizing a public key for the same user on different systems even belonging to different authentication realms or with mismatching user password: if public key authentication succeeds, access is indeed immediately granted.
- Since it is possible to authorize multiple keys for the same remote user, if your public key is among a remote user's authorized ones, this authentication mechanism also enables you to run a remote session as that other user.

During this kind of authentication, the client sends its public key to let the server verify the client's identity through a challenge-response mechanism.

If the key is not listed among the authorized ones for that specific user account (they must be stored in the `~/.ssh/authorized_keys` file), the server immediately terminates the connection.

If instead the public key is found among the authorized ones, the server generates an encrypted message and sends it back to the client.

If the client can decrypt the message, the session is started; otherwise the connection gets terminated.

Note Being able to decrypt the message requires having the related private key, and this is a proof of being the owner of the public key provided while authenticating. If instead the client cannot decrypt the message, the connection is terminated.

Generating SSH Authentication Key Pairs

To perform public key authentication, you must first generate an SSH key pair by running the `ssh-keygen` command-line utility.

In this example, we are generating the SSH key pair for the `mcarcano@carcano.corp` user:

```
ssh-keygen -b 2048 -C "mcarcano@carcano.corp"
```

Accept the proposed default file name for saving the generated key and, when asked, type the password you want to use for encrypting it.

Note Do not set an empty password, or the key will be saved unencrypted, and anybody who can access the file (e.g., running `sudo`) will be able to use the key and impersonate you.

The `-b 2048` option sets the key size to `2048` bits.

The default path of the generated keys is

- `~/.ssh/id_rsa` (encrypted private key)
- `~/.ssh/id_rsa.pub` (public key)

Authorizing a Public Key

Once you have generated your SSH key pair, you must add the public key to every SSH server you wish to connect to. This can be easily accomplished using the `ssh-copy-id` command-line tool.

Let's simulate this process using only the lab VM, pretending that `lab01.carcano.corp` is a remote host and that we are using the `ssh-copy-id` to authorize our public key on it.

While logged into the lab VM as your regular user, run the following command:

```
ssh-copy-id lab01.carcano.corp
```

This command establishes a password-based SSH connection to the `lab01.carcano.corp` host using the same user account you are currently logged in with. It then adds the contents of `~/.ssh/id_rsa.pub` to the `~/.ssh/authorized_keys` file on the remote server, unless it is already present.

Note Many SSH servers default configuration disables SSH access for the `root` user, so do not run this command as `root`. Additionally, if the user account you are using still does not have a password (or if you have forgotten it), you will need to set it or authentication will fail, for example, by running `sudo passwd` followed by the username you want to set the password.

By now it is possible to SSH connect to `lab01.carcano.corp` using public key authentication by simply running

```
ssh lab01.carcano.corp
```

You will be prompted to enter the password to unlock the encrypted private key located at `~/.ssh/id_rsa`.

Once connected, since this lab is complete, disconnect immediately by typing `exit` or pressing `CTRL+D`.

Each line in the `authorized_keys` file, in addition to the user's public key, may also contain additional directives for configuring restrictions such as limiting the source IP address granted to use that key, or an application to be automatically launched right after passing authentication, or other actions used to fine-grain permissions to reduce the attack surface.

For example, to restrict access while using `mcarcano@carcano.corp` public key only from clients from the `192.168.66.0/24` subnet except host `192.168.66.1`

```
from="192.168.66.*,!192.168.66.1" ssh-rsa AAAAB3Nza... mcarcano@
carcano.corp
```

Note Mind the above directive affects only public key authentication using that specific public key: any user connecting using password-based authentication, as well as using another authorized public key will not be subjected to the source IP restriction.

A real-life example is the way used by Gitolite to identify the user performing `git push` or `git pull`.

Note Gitolite is a command-line tool used to manage Git repositories with a strong focus on access control and permissions. When used in conjunction with Gitweb, the bundle can provide a basic web interface for viewing repositories, but this setup is limited and considered primitive compared to modern SCM systems. The bundle does not offer the full feature set of systems like Gitea, which includes collaboration tools, issue tracking, and a full-featured user-friendly web interface.

Gitolite requires authorizing multiple users' SSH public keys beneath the same `authorized_keys` file – for example, that of the `git` user. Since every push happens by SSH-connecting as the `git` user, to detect the Gitolite user pushing the changes, it requires prepending the `command` directive to each single line of the `authorized_keys` file, specifying to run the `/usr/bin/gitolite` command followed by the username that specific public key belongs to.

For example:

```
command="/usr/bin/gitolite mcarcano" ssh-rsa AAAAB3Nza... mcarcano@
carcano.corp
```

More info on `authorized_keys` is available on Linux Man Pages Online at https://www.ssh.com/academy/ssh/authorized-keys-openssh.

Note In real-life scenarios, you typically use an automation tool to distribute your public keys on every system you are granted access to. Even better, you may attach your public key to your account DN in the LDAP server and have the SSH servers set to automatically fetch users' public keys performing LDAP lookups. This latter tecnique typically requires developing a custom script.

The SSH Agent

Having to interactively enter the password of the encrypted private key at every connection quickly becomes disappointing, since it causes distractions and latency on the actual tasks you are instead focused on. In addition to that, this also causes troubles with batches, since by design they cannot be interactive.

To avoid this, you can set up the SSH Agent, which takes care of automatically unlocking the private key as needed.

Launching the Agent

There are generally two ways to launch `ssh-agent`: within a sub-shell or in the current shell (single shell).

In my personal experience, I prefer the single shell method because if the sub-shell crashes or is terminated, it can affect any processes running within it, including the `ssh-agent`.

Therefore, I will focus on demonstrating the single shell method.

Since when launched the `ssh-agent` prints the values of the `SSH_AUTH_SOCK` and `SSH_AGENT_PID` environment variables formatted with the syntax used to set shell variables, we can simply have the `eval` command to process them.

```
eval $(ssh-agent)
```

This way, the `SSH_AUTH_SOCK` and `SSH_AGENT_PID` variables are automatically exported in the current shell session.

Note The SSH_AUTH_SOCK variable contains the path to the UNIX domain socket to connect to the agent: the `ssh` client reads this variable to know if there's a running agent and how to forward requests to it. Mind that this is not limited to the `ssh` command-line client, but to any application leveraging the SSH protocol suite. This encompasses `git`, for example, while running `git push` and `git pull` if the repository were cloned using SSH as the transport protocol, Ansible statements, and so on.

Adding a Key

Once the agent is launched, we must add to it one or more private keys as necessary. For security reasons, the best practice is to always set a timeout, so as to have the key automatically removed once the timeout expires.

In the following example, we are setting a one-hour-only timeout (`-t 3600`):

```
ssh-add -t 3600 ~/.ssh/id_rsa
```

Obviously, it prompts you to type the decrypting password.

By now, connections to servers where the public key related to `~/.ssh/id_rsa` has already been authorized will no longer ask for any password.

Note Whenever the SSH server requests key-based authentication, the SSH client forwards this request to the `ssh-agent` process through a local UNIX socket. This design ensures that the client never directly sees the private key, enhancing security. This approach is reminiscent of the PKCS#11 token framework, where keys are managed and accessed only by the hardware token, keeping them secure from direct exposure. Additionally, `ssh-agent` supports PKCS#11 devices, allowing integration with hardware security modules (HSMs) and smart cards.

Removing a Key

In the same way as you added a private key, when it is no longer necessary, you can (actually you must) remove it by using the `-d` option. For example:

```
ssh-add -d ~/.ssh/id_rsa
```

Client Configuration Files

Note This section explains some configuration for the SSH client: before modifying any configuration files, please make a backup copy. You will need to restore the original files at the end of this chapter because, in the next chapter, we will be working with Ansible, which requires SSH to be properly configured.

The SSH client reads its configuration from a config file: the system-wide one is located at `/etc/ssh/ssh_config`. If needed, its settings can be overridden on a per-user basis by creating a `~/.ssh/config` file with only the settings you want to override.

Starting with Red Hat Enterprise Linux 9, the system also utilizes the `/etc/ssh/ssh_config.d` directory as an include directory for storing configuration snippets, allowing for more flexible configuration management.

For example, the new `/etc/ssh/ssh_config.d/50-redhat.conf` file, introduced to facilitate easier management of SSH settings, includes configurations related to cryptographic policies (we mentioned them in Chapter 25) and other SSH settings, streamlining the process for system administrators.

Note The documentation for the syntax of the configuration files can be easily retrieved by running `man ssh_config`.

The configuration files are structured in stanzas where settings are grouped by matching target hosts: every configuration stanza begins with the `Host` keyword followed by the matching rule.

The matching rule can either be for a specific host or for a group of hosts. In this last case we must use the * wildcard operator or the `!` negation operator to configure wider matching.

The following example provides a fictional example of a user's personal configuration file:

```
Host *.p1.carcano.corp !jump.p1.carcano.corp
    IdentityFile ~/.ssh/marco_carcano_id.rsa
    ProxyJump jump.p1.carcano.corp

Host *.t1.carcano.corp !jump.t1.carcano.corp
    IdentityFile ~/.ssh/marco_carcano_id.rsa
    ProxyJump jump.t1.carcano.corp

Host *.l1.carcano.corp !jump.l1.carcano.corp
    IdentityFile ~/.ssh/marco_carcano_lab_id.rsa
    VerifyHostKeyDNS=no
    StrictHostKeyChecking=no
    ProxyJump jump.l1.carcano.corp
```

This has been designed for a scenario with three distinct environments:

- `l1` - Lab1: This is an environment used to manage disposable labs. Nearly every user has access to it, so it is the one with the weakest security level. For this reason, here we don't use the usual SSH key pair we use for the other environments: we are instead using a different key saved in the `~/.ssh/marco_carcano_lab_id.rsa` file. This way, if this key gets compromised, the breach is confined to the `lab1` environment only. In this environment, we also disable Host Keys verification: from the security perspective it is pointless and enforcing it is just a waste of effort since the systems are expected to be frequently reprovisioned with new host keys.
- `t1` - Test1: A production-grade environment dedicated to continuous testing.
- `p1` - Production 1: The actual production environment.

The number `1` in each environment is the security tier: this enables having multiple parallel environments with different security policies.

It is important to remark on the use of the `ProxyJump` directive: this forces the connection to the matching hosts to be proxied through the host specified in this directive.

This configuration is typically used in scenarios where security enforces access to target hosts through bastion hosts.

In such a scenario, if using public key authentication, the public key must be authorized on the jump host as well: you can do it manually by appending the public key to the jump host or using the `ssh-copy-id` command-line utility.

Note The settings in these configuration files affect not only connections performed by the `ssh` client command-line utility, but every connection using SSH as transport. This includes, for example, Git, Ansible, and so on.

Connecting to a Server

Connections to a server are performed using the `ssh` client command-line utility.

Let's illustrate it by using a few fictional use cases.

The most straightforward invocation requires providing only the target host, for example:

```
ssh mysql-ul1a001.l1.carcano.org
```

By default, SSH connects to the remote host using the same username connected to the current session. If you need to use a different one, you must prepend it to the host you are connecting to, along with a trailing `@` character. For example:

```
ssh otheruser@mysql-ul1a001.l1.carcano.org
```

Another way to provide the remote user is using the `-l` command-line option. For example:

```
ssh -l otheruser mysql-ul1a001.l1.carcano.org
```

Other commonly used `ssh` command-line options are shown in Table 36-1.

Table 36-1. *ssh client command-line options*

Option	Description

`-i`	Specify or override the identity file to use, which contains the private key. For example:
	`ssh -i /var/lib/fooservice/.ssh/id.rsa \`
	`-l fooservice apps-up1a001.p1.carcano.org`

`-p`	Specify the port to connect to: this is used to connect to SSH services not listening on the default port (TCP/22). For example: `ssh -p 2222 git-up1a001.p1.carcano.org`

When necessary, it is possible to override configurations on a per-session basis by adding the relevant SSH option to the `ssh` connection statement using the `-o` command-line option.

Note The available SSH options are the same available for the configuration files.

For example:

```
ssh -o VerifyHostKeyDNS=no -o StrictHostKeyChecking=no \
mysql-up1a001.p1.carcano.org
```

Connect to the `mysql-up1a001.p1.carcano.org` server, preventing the DNS lookup for the `SSHFP` records of the servers' FQDN or IP address, and disable the server's host key validation (this causes any host key's fingerprint to be automatically added to the `~/.ssh/known_hosts` file).

Note Be careful with these two options: when using them you are vulnerable to Man-in-the-Middle attacks.

Executing Remote Commands

SSH enables also running remote commands on the SSH server: it is enough to specify the statement to run right after the SSH connection statement.

For example:

```
ssh lab01.carcano.corp /usr/bin/who
```

Runs the `/usr/bin/who` command on the `lab01.carcano.corp` remote SSH server.

This feature is particularly useful when writing automation batches to prepare environments spanning multiple hosts, but it can be exploited to address many other use cases.

As a pearl, I want to share with you a trick I often use to perform live capture using TCPdump on a remote host while running Wireshark locally on my Apple MAC - mind it requires the remote user to have sudo rights on the remote host:

```
ssh fancyhost.carcano.corp sudo tcpdump -U -s0 'tcp port 3000' -i any -w -| 
Wireshark -k -i -
```

The above statement, after connecting to the `fancyhost.carcano.corp` remote host, runs the `tcpdump` command-line utility as the `root` user, invoking it to capture from any network interface `TCP` traffic on port `3000`. Then, it launches `Wireshark` on the client, piping the tcpdump output to it.

Note Wireshark is a free and open-source network packet analyzer used for network troubleshooting and analysis. It captures and displays packet data in detail, allowing users to inspect network traffic and diagnose issues. Wireshark is widely utilized in software and communications protocol development, so it is a tool I recommend you familiarize yourself with. It provides a graphical interface for analyzing network packets, enabling users to view and dissect packet data across various OSI layers. It supports extensive filtering and statistical tools for in-depth network analysis. This tool is essential for network administrators and security professionals to monitor and troubleshoot network performance.

Secure Copy – SCP

As we discussed, the SSH suite has been specifically developed to provide secure alternatives to dangerous utilities such as `rcp`. The SSH equivalent command for `rcp` is `scp`, which is a command-line tool that enables you to securely copy files between hosts.

Note Although `scp` is still broadly used, it is deprecated in Red Hat Enterprise Linux (RHEL) 9. Despite relying on the SFTP subsystem, the `scp` API has known security vulnerabilities that have led to its deprecation in the `libssh` library. While the `scp` command remains available in RHEL 9, it is recommended to start using SFTP for file transfers before its deprecation is finalized. In this book, we mention `scp` only for the sake of completeness.

The syntax of the command is quite simple:

```
scp source destination
```

Within the `source` or `destination` path, you can also specify the FQDN of remote hosts as needed.

For example, to copy the local file `/usr/share/doc/openssh/README` to the `/tmp` directory of the remote host `apps-up1a001.p1.carcano.org`, you would use::

```
scp /usr/share/doc/openssh/README \
apps-up1a001.p1.carcano.org:/tmp
```

You can, of course, do the opposite, copying the file `/usr/share/doc/openssh/README` from the remote host `apps-up1a001.p1.carcano.org` to the `/tmp` directory of the local system:

```
scp \
apps-up1a001.p1.carcano.org:/usr/share/doc/openssh/README \
tmp
```

If necessary, you can also recurse across subdirectories by providing the `-r` option, but be very careful with this: `scp` will copy links (both symlinks and hard links) as files; the risk is to fall into infinite loops if there are any circular directories.

If you need to preserve the permissions of the copied contents, just use the `-p` command-line option.

As for globbing, remember that escaping is done first by the running shell and then by the command. This means that, for example, the following will not work:

```
scp apps-up1a001.p1.carcano.org:/home/marco/*.txt /tmp
```

unless you escape it as follows:

```
scp apps-up1a001.p1.carcano.org:/home/marco/\*.txt /tmp
```

SFTP

From an operational perspective, the `sftp` client works pretty similar to the regular FTP: you just need to connect to the SFTP host and then run the usual FTP commands (`cd`, `ls`, `get`, `put`, `exit`, ...).

Connecting just requires specifying the target SFTP host. For example:

```
sftp sftp.p1.carcano.org
```

The man page of the `sftp` command-line tool provides extensive documentation with all the available commands, so if you want to know more, just type `man sftp`.

A Few SSH Server Security Best Practices

Note This section covers some security tweaks for the SSH daemon (SSHd) server. Before modifying any configuration files, please make a backup copy. You will need to restore the original files at the end of this chapter because, in the next chapter, we will be working with Ansible, which requires the SSH server to be properly configured.

As DevSecOps professionals, we must ensure the implementation of basic security for the SSH service.

The SSH daemon (SSHd) service settings are configured in the `/etc/ssh/sshd_config` file. Starting with Red Hat Enterprise Linux 9, the system also utilizes the `/etc/ssh/sshd_config.d` directory as an include directory for storing configuration snippets, allowing for more flexible configuration management.

For example, the new `/etc/ssh/sshd_config.d/50-redhat.conf` file, introduced to facilitate easier management of SSH settings, includes configurations related to cryptographic policies and other SSH settings, streamlining the process for system administrators.

Be aware that any changes made to the main configuration file or to the included snippets require restarting the SSH daemon by running

```
sudo systemctl restart sshd
```

The most straightforward security best practice is reducing the attack surface by restricting SSH access to selected users and groups. This is managed by the directives `AllowUsers`, `DenyUsers`, `AllowGroups`, and `DenyGroups`.

For example, to grant access only to the `vagrant` and `practitioner` users:

```
AllowUsers vagrant practitioner
```

Or, to grant access only to the `vagrant` and `devops` groups:

```
AllowGroups vagrant devops
```

Note When dealing with a VM deployed with Vagrant, you must always grant access to the `vagrant` user or the `vagrant` group.

It is important to remark that the allow/deny directives are processed in the following order:

```
DenyUsers, AllowUsers, DenyGroups, AllowGroups
```

This has the clear drawback that, when using `AllowGroups`, if `AllowUsers` is specified, any user member of the supplied group not listed in `AllowUsers` is denied access to connect.

Note In real life setups, authentication and authorization are very often managed by the `sssd` service, which enables configuring LDAP and Kerberos authentication. In such scenarios, access authorization is generally configured at the `sssd` level.

Table 36-2 summarizes the most commonly used security-related SSHd options.

Table 36-2. *SSHd most used security options*

Option	Description
`PasswordAuthentication`	Enable and disable password-based authentication. For example, to disable it `PasswordAuthentication no`
`PermitRootLogin no`	Permit or forbid the `root` user to login to the system. Security best practices require you to disable it by setting `PermitRootLogin no`
`AllowTcpForwarding`	TCP forwarding enables the user to use the SSH server as a jump host to forward connections to any other host reachable from it. This obviously increases security risks. Unless the purpose of the host is working as a jump host, you must disable TCP forwarding as follows: `AllowTcpForwarding no`
`X11Forwarding`	X11 forwarding enables users to redirect X11 applications to remote display servers. Since this is required only in a very few scenarios, unless you really need this, you must disable it as follows: `X11Forwarding no`
`PermitTunnel`	Permit or forbid creating SSH tunnels. From the security perspective, this is always a huge security hole. You must disable this by setting: `PermitTunnel no`

Note Before moving to the next chapter, if you made any changes to the SSHd server or client configuration, please restore the original versions from the backup files you created.

Summary

This chapter provided an overview of SSH, focusing on operational concepts such as handling host keys, utilizing password-based and public key authentication, and employing the `ssh-agent` for automatic key unlocking. We also explored SSH client configuration and server security settings.

For DevOps and DevSecOps professionals, proficiency in Secure Shell is essential for secure communication and management of remote systems:

- Understanding host keys is critical, as using incorrect keys can lead to significant security risks, including unauthorized access.
- Effective management of authorized keys, including security measures like limiting source access and automating command execution, enables professionals to create robust configurations that address complex scenarios while avoiding security pitfalls.
- The `ssh-agent` plays a vital role in securely managing private keys, facilitating seamless authentication, and enabling the secure execution of scripts involving SSH commands, such as remote command execution or file transfers using SCP or SFTP.
- A thorough understanding of client and server options is crucial for fine-tuning permissions and tightening security, which helps reduce the attack surface.

Overall, these skills enhance operational efficiency and significantly strengthen the security posture of the infrastructure managed by DevOps and DevSecOps teams.

CHAPTER 37

Ansible Playbooks, Roles, and Collections

Ansible is a powerful datacenter automation tool that enables nearly declarative automations.

It is so broadly used that being skilled on it is a must for DevSecOps professionals: knowing how to use it enable to quickly and reliably deliver infrastructures and applications using an imperative approach that, with a proper design of playbooks and modules, can even become nearly declarative, using manifests to describe the topology, structure, and configuration of each deployment.

Its idempotent design not only enables redeploying at will failed deployments without worrying about starting from the failure point: it is also useful to provide consistency checks of every deployment, fixing inconsistencies on the fly.

Moreover, it also has a check-only feature that enables runs without applying changes, reporting only what would be done in a real run.

This chapter is a primer with Ansible (thoroughly describing it would require a whole book), gradually introducing concepts, providing a good baseline that quickly enables operating Ansible running ad hoc statements, playbooks and operating using Ansible Galaxy with off-the-shelf roles and collections.

To avoid being distracted by setting up the operating environment before each specific example, we set up a baseline lab right now.

In real life, sysadmin users run Ansible having it connect to the target system with a dedicated system user. This of course requires creating that system user on the target systems.

The most convenient way to do that is using a dedicated Ansible playbook to prepare target hosts not only creating the dedicated system user, but also all the other necessary settings (such as a `sudo` rule dedicated to it). When performing this task, the sysadmins run such a playbook using their own corporate user or, if they have it, a dedicated sudo-enabled service user previously set up for other means.

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9_37

To simulate this in our lab, we must make sure the user you are using has the same privileges as a sysadmin. In other words, it must be granted to run `sudo` statements. We discussed how to configure `sudo` rules in Chapter 21, so you should now be able to perform this configuration.

In this chapter, since I am assuming to be a sysadmin using `mcarcano` as my personal user, I add the `/etc/sudoers.d/mcarcano` file with the following contents:

```
mcarcano ALL=(ALL) NOPASSWD: ALL
```

You must of course rework the above rule to match the username you chose.

Note Before going any further, we must change to this **chapter's workspace directory**. Just run `cd ~/devsecops-pillars/ch37`.

Containerized Ansible

Ansible tends to require bleeding edge Python features not available on enterprise class Linux distributions: despite it being possible to install them using the Python virtual environment, it winds up quite quickly into a mess of Python versions that, even if you install it at the user level, will dirty your system.

In my working experience the most convenient way to use Ansible is running it inside a container image – this approach provides lots of benefits:

- It demands very little setup and maintenance effort (you don't need to install or patch it at all).
- You can very easily switch between different Ansible versions – it is just a matter of specifying the container image you want to run.
- It provides an out-of-the-box way to always have the development and the operational environments aligned – it is just a matter of running the same container image.
- It is very easy not only to integrate within an existing CI/CD suite, but also to migrate to a different CI/CD suite when it is necessary.

Set Up the Container Environment

Since we want to use containerized Ansible, unless it is already installed, we must install Docker as described in Chapter 28.

Pull the Ansible Container

To speed up the Ansible container initialization, we can prefetch the container image by using `docker pull`. For example:

```
docker pull docker.io/alpine/ansible:latest
```

Set Up the Ansible Directory Tree

The current chapter's workspace directory contains the `ansible` subdirectory with some resources we will need while running our labs. Let's have a look at its contents.

```
ls ansible
```

This directory already contains the `playbooks` directory with some example playbooks we will run in our labs.

To complete the setup of the Ansible environment, we are still missing the following directories:

- `environment`: used to store the Ansible environment
- `roles`: used to store the Ansible roles
- `collections`: used to store the Ansible collections

Let's create them now as follows:

```
mkdir -m 755 ansible/environment ansible/roles \
ansible/collections
```

Configure Ansible

Ansible reads its settings from the `ansible.cfg` configuration file.

We can generate it by launching the Ansible container as follows:

```
docker run --rm -it -w /ansible -v $(pwd)/ansible:/ansible:Z \
docker.io/alpine/ansible:latest \
bash -c "ansible-config init --disabled > ansible.cfg"
```

The above statement spun up a container and

- Bind-mounted the `/ansible` directory inside the container to the local `ansible` directory (`$(pwd)/ansible:/ansible:Z`)
- Used the `/ansible` directory inside the container as the working directory (`-w /ansible`) while running the `ansible-config` command
- Ran the `ansible-config` command-line tool to generate the default `ansible.cfg` configuration file in the `/ansible` directory within the container

Since the `/ansible` directory in the container was bind-mounted to the `ansible` directory on the host (our lab VM), then the `ansible.cfg` configuration file is actually stored beneath the `ansible` directory.

Note In the rest of this chapter there are several statements having to deal with contents of the `ansible` directory: the actual path will depend on the scope (container or host) while running the statement, so please pay attention to it.

We can verify that the `ansible.cfg` file is actually in the VM's filesystem as follows:

```
ls -al ansible/ansible.cfg
```

As you can see (and should expect), it is owned by the `root` user.

For our convenience, we can take ownership by running

```
sudo chown $(whoami): ansible/ansible.cfg
```

Let's now adjust the generated file to suit our needs.

First, we configure Ansible to login to the target hosts as the `ansible` user:

```
sed -i 's/^[ ]*[;][ ]*remote_user[ ]*=.*/remote_user=ansible/' ansible/
ansible.cfg
```

Since this book is focused on Linux, in this chapter I assume only Linux target hosts, but be wary Ansible supports also Microsoft Windows target hosts.

Ansible connects to Linux hosts using the so-called `smart` connection, which relies on SSH. Although Ansible supports password-based SSH logins to the target hosts, the most convenient way for connecting is of course using public key authentication.

Let's configure Ansible to use the `environment/ansible.key` SSH private key:

```
sed -i 's/^[ ]*[;][ ]*private_key_file[ ]*=.*/private_key_file=
environment\/ansible.key/' ansible/ansible.cfg
```

Since the private key still does not exist, we must generate it. Please mind that, as we are on the host, the path where to save it is `ansible/environment/ansible.key`:

```
ssh-keygen -t rsa -b 4096 -f ansible/environment/ansible.key
```

Pick up a good password and enter it when requested.

We must then rename `ansible/environment/ansible.key.pub` into `ansible/environment/ansible.pub`:

```
mv ansible/environment/ansible.key.pub \
ansible/environment/ansible.pub
```

and also create the `~/.ssh/known_hosts` file if it does not exist yet.

```
[ -f ~/.ssh/known_hosts ] || touch -f ~/.ssh/known_hosts
```

Ansible needs an inventory where the target hosts are listed, grouped, and described by providing attributes – in this example we configure it to use the `environment/hosts` file:

```
sed -i 's/^[ ]*[;][ ]*inventory[ ]*=.*/inventory=environment\/hosts/'
ansible/ansible.cfg
```

Let's then create the `ansible/environment/hosts` – just run

```
cat << EOF > ansible/environment/hosts
localhost ansible_connection=local

[workshop]
${HOSTNAME} host_labels=[postgresql]
EOF
```

Let's have a look to the contents by running `cat ansible/environment/hosts` :

```
localhost ansible_connection=local

[workshop]
lab01.carcano.corp host_labels=[postgresql]
```

This is a very minimal inventory suitable for this chapter's purposes.

It just contains only the following target hosts:

- `localhost`
- `lab01.carcano.corp`

Only for the `localhost` target host, it also sets the `ansible_connection` variable to `local` – this override avoids Ansible from authenticating or connecting when operating on the container itself, for example, to install missing APK packages.

Since we will need this in some labs, the `lab01.carcano.corp` belongs to the `workshop` group and has the `host_labels` variable of type list with `postgresql` as the only item.

We must then configure Ansible to look in the `roles` directory for the Ansible roles to load

```
sed -i 's/^[ ]*[;][ ]*roles_path[ ]*=.*/roles_path=roles/' ansible/
ansible.cfg
```

and in the `collections` directory for the Ansible collections to load

```
sed -i 's/^[ ]*[;][ ]*collections_path[ ]*=.*/collections_
path=collections/' ansible/ansible.cfg
```

We will discuss Ansible roles and collections later on in this chapter.

Run Ansible Within the Container

We are now ready for a go with the containerized Ansible. As an example, let's add the message `System managed by Ansible` to the `/etc/issue.net` file on the `lab01.carcano.corp` host by running an Ansible ad hoc statement.

Note All the following examples assume Ansible connects to the `lab01.carcano.corp` host as `mcarcano` user. You must change it to fit your scenario, replacing `mcarcano` with the username you are using on your system.

```
docker run --rm -it --network=host -w /ansible \
-v $(pwd)/ansible:/ansible:Z \
-v $(pwd)/.ssh/known_hosts:/root/.ssh/known_hosts:Z \
docker.io/alpine/ansible:latest \
bash -c "apk add sshpass; ansible -b -u mcarcano -k lab01.carcano.corp \
--ssh-extra-args='-o StrictHostKeyChecking=no' \
-m copy -a \"content='System managed by Ansible\n' dest=/etc/issue.net\""
```

The above statement spun up a container and

- Bind-mounted the `/ansible` directory inside the container to the local `ansible` directory (`$(pwd)/ansible:/ansible:Z`)
- Used the `/ansible` directory inside the container as the working directory (`-w /ansible`)
- Ran the `apk add sshpass` statement: this step is necessary to enable Ansible to connect to the target hosts using password authentication. Of course, if the user you are using to connect to the target system has already authorized your public key, you can of course omit it. When using public key authentication, it is not necessary to provide the `-k` Ansible's command-line parameter.
- Ran the actual Ansible ad hoc statement, invoking the `copy` module passing the following arguments (`-a` option):
  - `content='System managed by Ansible\n'`
  - `dest=/etc/issue.net`

  The statement contains many other Ansible's parameters which we will explain shortly when talking about Ansible ad hoc statements.

Ansible connects using SSH to the target host.

Right after entering the password of the `mcarcano` user on `lab01.carcano.corp`, if everything is properly working, it prints the following output:

```
lab01.carcano.corp | CHANGED => {
      "ansible_facts": {
      "discovered_interpreter_python": "/usr/bin/python3.9"
      },
      "changed": true,
      "checksum": "609234201def229bc830278f5018bb8b640cc123",
      "dest": "/etc/issue.net",
      "gid": 0,
      "group": "root",
      "md5sum": "948458ccc530da949b0f2dc515b6b5a8",
      "mode": "0644",
      "owner": "root",
      "size": 32,
      "src": "/home/mcarcano/.ansible/tmp/ansible-tmp-1748526823.462444-
      10-245232182480205/.source.net",
      "state": "file",
      "uid": 0
}
```

The `CHANGED` word at the very beginning means the module applied a change – so, in this specific case, the `/etc/issue.net` file now contains the message `System managed by Ansible`.

Launching the Containerized Ansible Directly from the Shell

Yes, running Ansible like that is very annoying, since it requires providing the `docker` statement with lots of arguments, and also paying attention to properly escaping Ansible's parameters too.

The solution to this problem is to create a shell function, so that you can launch as if it was the actual `ansible` command-line tool installed on the system.

The `ansible-container.sh` in the chapter's workspace directory contains the definition of the functions `ansible`, `ansible-playbook`, `ansible-galaxy`, `ansible-inventory` and `ansible-vault`.

Note Spend some time checking it – at this point of the book you already have the skills to make sense of everything on your own. For clarity, the functions contain a statement to install the Alpine's `py3-netaddr` APK package because it is a mandatory requirement for being able to use filters such as `ipaddr`. You can of course build your own ansible container image, deriving it from the official Alpine one, encompassing the `py3-netaddr` APK package and any other package you may need: in this case, you can safely remove the statements which install APK packages from the functions contained in the `ansible-container.sh` file,

---

To have these functions automatically loaded by every user at login time, just copy the `ansible-container.sh` file to the `/etc/profile.d` directory.

```
sudo cp ansible-container.sh /etc/profile.d
```

Disconnect and reconnect your session, or source the `/etc/profile.d/ansible-container.sh` to load the functions.

Note Once reconnected, remember to navigate to the `~/devsecops-pillars/ch37` directory.

Let's test the functions we just installed to see if they are properly aliasing the `ansible` command-line tools.

For example, let's run the `ansible-inventory` command:

```
ansible-inventory -i /ansible/environment/hosts --vars --graph
```

The output is

```
@all:
  |--@ungrouped:
  |  |--localhost
  |  |  |--{ansible_connection = local}
```

```
|--@workshop:
|  |--lab01.carcano.corp
|  |  |--{host_labels = [postgresql]}
```

Ansible Ad Hoc Statements

Ansible leverages its own idempotent Python modules, providing an easy way to inject them into the target hosts as temporary files and run them.

The most basic way for running Ansible is the so-called ad hoc mode: in this way Ansible connects to the remote target and runs just the specified module with the supplied arguments.

We already have run an ad hoc statement to set the `System managed by Ansible` message in the `/etc/issue.net` file, but that required running the full `docker` statement since we hadn't added the function to emulate the regular `ansible` command in `/etc/profile.d/ansible-container.sh`.

Let's re-run it, this time as a regular `ansible` statement:

```
ansible -b -u mcarcano -k lab01.carcano.corp \
--ssh-extra-args='-o StrictHostKeyChecking=no' \
-m copy -a "content='System managed by Ansible\n' \
dest=/etc/issue.net"
```

The above statement runs the actual Ansible ad hoc statement, invoking the `copy` module.

We provided also Ansible's command-line tool parameters as follows:

- `-u mcarcano`: Tells Ansible to connect to the target system as the `mcarcano` user
- `-k`: Tells Ansible to prompt for the user's password
- `-b`: Stands for `become`, which in Ansible terms means right after connecting to the target system, become another user (the `root` user by default)
- `--ssh-extra-args='-o StrictHostKeyChecking=no'`: Disables the check of the target host's host key

As you can see, this time we used the same syntax we would use if Ansible was actually installed on our workstation.

This time the output looks as follows:

```
lab01.carcano.corp | SUCCESS => {
      "ansible_facts": {
      "discovered_interpreter_python": "/usr/bin/python3.9"
      },
      "changed": false,
      "checksum": "609234201def229bc830278f5018bb8b640cc123",
      "dest": "/etc/issue.net",
      "gid": 0,
      "group": "root",
      "mode": "0644",
      "owner": "root",
      "path": "/etc/issue.net",
      "size": 32,
      "state": "file",
      "uid": 0
}
```

The `SUCCESS` word at the very beginning means the module was successfully run, but didn't apply any change since the target already matches the expected outcome.

This is one of Ansible's key features: idempotency, which means being able to re-run the same statement multiple times generating the same outcome. In addition to that, it provides valuable information about what has changed and what was already OK; that is very useful even just for running compliance checks.

If you studied the contents of the `/etc/profile.d/ansible-container.sh` file, you certainly know that it was specifically designed to enable you to specify the Ansible container version to use by setting the `ANSIBLE_CONTAINER_LABEL` variable.

For example, set the `ANSIBLE_CONTAINER_LABEL` variable as follows:

```
ANSIBLE_CONTAINER_LABEL=2.18.1
```

Now, every next `ansible` statement will use the `alpine/ansible:2.18.1` container image.

Let's try this by completing the change we were doing, configuring SSHD to use the `/etc/issue.net` file we previously copied with the Ansible ad hoc statement as the SSH banner file. This requires adding the `Banner /etc/issue.net` directive to the `/etc/ssh/sshd_config` file – just run

```
ansible -b -u mcarcano -k lab01.carcano.corp \
--ssh-extra-args='-o StrictHostKeyChecking=no' \
-m lineinfile -a "path=/etc/ssh/sshd_config regexp='^[#][ ]*Banner .*'
line='Banner /etc/issue.net'"
```

The above statement runs the actual Ansible ad hoc statement, invoking the `lineinfile` module (`-m` option) passing the following arguments (`-a` option):

- `path=/etc/ssh/sshd_config`: the file to change
- `regexp='^[#][ ]*Banner .*'`: the line to match
- `line='Banner /etc/issue.net'`: the changed text

This time the output message is

```
lab01.carcano.corp | CHANGED => {
      "ansible_facts": {
      "discovered_interpreter_python": "/usr/bin/python3.9"
      },
      "backup": "",
      "changed": true,
      "msg": "line replaced"
}
```

So, this time the `lineinfile` Ansible module applied the change.

Lastly, we must restart the SSHD service to apply the change:

```
ansible -b -u mcarcano -k lab01.carcano.corp \
--ssh-extra-args='-o StrictHostKeyChecking=no' \
-m service -a "name=sshd state=restarted"
```

Since the output message is very long, I don't put it here to save space: the only important thing to remark is that also the `service` Ansible module reports `CHANGED` meaning the `sshd` service has been restarted.

Note You are encouraged to have a look at all the most used built-in Ansible modules: they are documented at https://docs.ansible.com/ansible/latest/collections/ansible/builtin/.

To see the outcome of change in action, just try to SSH connect to the `lab01.carcano.corp` host:

```
ssh lab01.carcano.corp
```

Right before asking you to enter the password, it must show the following message:

```
System managed by Ansible
```

Disconnect the SSH connection by typing `exit` or pressing `CTRL+D`.

Ansible Playbooks

As you can see Ansible is a powerful tool, but if the only way to run Ansible statements was the ad hoc mode, Ansible wouldn't be a very handy and popular tool.

Note Ad hoc is a way for using Ansible most suitable when having to deal with single statements to run massively, such as a mass restart of services or target systems.

When having to deal with a large list of tasks, Ansible must be run using playbooks.

These are YAML-formatted files containing lists of tasks to run, either sequentially or on specific events (these are called handlers).

These tasks are grouped into what is called `play`. Since these files can actually contain more than just one play, they are called `playbooks`.

An Ansible play is not just a list of tasks specifying modules and their settings: it has its own syntax, supporting control structures such as loop and conditionals. In addition to that, it supports the JINJA2 templating language.

Playbooks are processed by Ansible using the `ansible-playbook` command-line utility.

Writing an Ansible Playbook

This chapter is just a primer with Ansible, so we are about to see a quite basic playbook (it's missing control structures and conditionals).

The most immediate use case to address is "preparing" target hosts for being managed by Ansible: to illustrate how to address this use case using Ansible itself, the book's companion resources provide the `ansible/playbooks/engine/join.yml` playbook.

It contains the following two plays:

1. `Prepare the Ansible container`
2. `Prepare targets for being managed by Ansible`

The definition of the `prepare the Ansible Container` play is as follows:

```
- name: prepare the Ansible Container
  hosts: localhost
  become: true
  tasks:
    - name: Update repositories and install the sshpass
      community.general.apk:
        name: sshpass
        update_cache: true
```

This play runs on the `localhost` target, which is the Ansible container itself, and contains a single task for installing the `sshpass` APK package using the `community.general.apk` Ansible module.

Note The `community.general.apk` Ansible module is used to install APK packages.

The `sshpass` APK package is used to enable `ansible` statements to use password-based SSH authentication: since it is very rarely required when running `ansible-playbook` statements, the `ansible-playbook` function defined in `/etc/profile.d/ansible-container.sh` does not automatically install it.

Conversely, preparing an Ansible target is a corner case requiring password-based SSH authentication. Within this playbook, Ansible performs the initial SSH connection using password-based authentication and switches to public key authentication only after having authorized the Ansible's system user public key.

Let's dig into the `prepare targets for being managed by Ansible` play.

```
- name: prepare targets for being managed by Ansible
  hosts: "{{ targets }}"
  vars:
    ansible_host_key_checking: false
```

The above snippet sets as target host any value of the `targets` variable and disables SSH host key checking – this is necessary since this is the first SSH connection attempt and so it is expected to have to deal with unknown SSH host keys.

Note After the first connection the host key is automatically added among the `~.ssh/known_hosts` file, so this option is not needed in any other playbook.

The definition of the `play` then continues with the `tasks` list:

```
  tasks:
    - name: create the Ansible service user
      become: true
      ansible.builtin.user:
        name: "{{ ansible_svc_user }}"
```

This first task, as the name suggests, creates Ansible's system user by running the `ansible.builtin.user` Ansible module. The actual created username is the one defined by the `ansible_svc_user` variable.

```
    - name: setting the  Ansible service user's password
      become: true
      ansible.builtin.shell: |
        echo "{{ ansible_svc_user }}:{{ ansible_svc_password }}" | chpasswd
        -c SHA512
```

The above snippet defines the second task: it invokes the `ansible.builtin.shell` Ansible module to run the `chpasswd` command-line utility to set the value of the `ansible_svc_password` variable as password for Ansible system user.

```
- name: set the Ansible service user's public key
  become_user: "{{ ansible_svc_user }}"
  become: true
  ansible.builtin.authorized_key:
    user: "{{ ansible_svc_user }}"
    key: "{{lookup('file', '/ansible/environment/ansible.pub')}}"
```

This third task runs after switching identity to the Ansible: the user switching is invoked by the `become: true` directive, whereas the user to impersonate is specified by `become_user: "{{ ansible_svc_user }}"`.

It then runs the `ansible.builtin.authorized_key` Ansible module to add a key to the Ansible system user's `authorized_key` file: the key is loaded from the `/ansible/environment/ansible.pub` file in the Ansible container using the `lookup` function.

```
- name: grant sudo without password to the Ansible service
  become: true
  ansible.builtin.copy:
    content: "{{ ansible_svc_user }} ALL=(ALL)NOPASSWD: ALL"
    dest: /etc/sudoers.d/{{ ansible_svc_user }}
```

The above fourth task runs after switching identity to the `root` user (`become: true`): it runs the `ansible.builtin.copy` Ansible module to create a sudo rules file with the values of the `ansible_svc_user` variable as the name in the `/etc/sudoers.d` directory. The sudo rule enables the Ansible system user to run whatever statement as any user on any without being requested to enter a password.

Note Since this is a lab environment, it is safe to create the sudo rule with the `NOPASSWD` option. In real life scenarios you must avoid this and specify the `become` password every time you run the `ansible` and `ansible-playbook` commands.

```
  - name: making sure the Ansible service user's sudo grant
    become: true
    become_user: "{{ ansible_svc_user }}"
    ansible.builtin.command: "sudo -l"
```

The fifth and last task run is just used to test the sudo rule – after switching to the Ansible system user, it invokes the `ansible.builtin.command` Ansible module to run the `sudo -l` statement.

Note A golden rule when performing critical configuration changes is to always add additional tasks to check the system behaves properly after that check, automatically rolling back the change and aborting the play in case of failures.

Running a Playbook

Let's try having a go with the playbook. As discussed, this playbook requires to set a few Ansible variables, among which there is `ansible_svc_password` with the password to assign to the Ansible's system user.

To prevent the password from being captured by the shell history, we read it from the TTY and store it into the `ANSIBLE_SVC_PASSWORD` variable:

```
read -s ANSIBLE_SVC_PASSWORD
```

We are now ready for our first go with our Ansible playbook:

```
ansible-playbook -u mcarcano -k \
-e targets=lab01.carcano.corp -e ansible_svc_user=ansible \
-e ansible_svc_password=${ANSIBLE_SVC_PASSWORD} \
playbooks/engine/join.yml
```

Once launched, you will be prompted to enter the password of the user you specified for connecting to the `lab01.carcano.corp` host (`mcarcano` in this example).

The above statement runs the `ansible-playbook` command-line tool to

- Connect to the target systems as `mcarcano` user and prompt for its password (`-k` option)

- Configure the Ansible local system user to be created on the target host as `ansible` with the password defined in the `ANSIBLE_SVC_PASSWORD` variable
- Run the playbook only on the `lab01.carcano.corp` target host
- Run the `playbooks/engine/join.yml` playbook

Note In order to have Ansible's `facter` to be able to detect all the available facts on the target host, on Red Hat family target hosts it is necessary to install the `redhat-lsb-core` RPM package. The problem is that on derivative distros, such as Oracle Linux or Rocky Linux, that RPM package is provided by a repository that is not enabled by default. This means that it is necessary to enable it first, then install the package and lastly disable it again. Ansible manages the enabling and disabling of DNF repositories using the `dnf_config_manager` module available in the `community.general` Ansible collection. This is definitively a good use case for going on and seeing how to deal with shelf Ansible collections, so we'll see how to address it when talking about Ansible Collections.

Before continuing, remember to unset the `ANSIBLE_SVC_PASSWORD` variable:

```
unset ANSIBLE_SVC_PASSWORD
```

Ansible Namespaces

As we have seen so far, Ansible is a very modular system leveraging Python, modules, plugins, and playbooks.

A problem every modular system must address is avoiding naming collisions, such as having different entities writing modules or playbooks with the same name.

Ansible addresses the naming collision problem in the most common and trivial way: when invoking objects, such as modules or playbooks, outside of the ones provided by the Ansible distribution itself, it is mandatory to prepend them specifying their namespace: if the namespace part is omitted, then Ansible looks for the referenced object into the default namespace (`ansible.builtin`).

So, the syntax to be used when referring to objects is

```
<namespace>.<objectname>
```

Ansible Roles and Collections

Ansible of course fosters also the Don't Repeat Yourself (DRY) paradigm: it is indeed possible to just import tasks list from existing YAML files, so to be able to re-use them as necessary.

In addition to that, Ansible provides two delivery formats that go far beyond just the simple providing of reusable task lists.

Ansible Roles

Ansible roles are a convenient format enabling delivery within a single package:

- Reusable task list
- Resource files and templates
- Handlers
- Var files and defaults
- Custom Ansible modules

All of these are packaged within a gzipped tarball along with some metadata useful for determining its version, requirements (e.g., minimum supported Ansible version), and operating environment (e.g., the supported operating systems).

As you can easily guess, the best practice is to use namespaces, so you must include the name space when creating or invoking a role.

For example:

```
<namespace>.<role_name>
```

The only exception is Ansible roles embedded in collections, since the best practice is to have the collection itself already defined within a namespace.

Ansible Collections

Ansible collections go far beyond Ansible roles, enabling delivery within a single package:

- Playbooks
- Roles
- Var files
- Ansible modules
- Ansible plugins

This enabled third parties to start to develop and deliver their own Ansible contents, managing versioning at their own pace instead of at Ansible project's one, contributing to the wide-spreading of Ansible's adoption.

As you can easily infer, the namespacing rule applies also to Ansible collections, requiring their names to be prepended by the namespace they belong to.

Invoking objects inside a collection requires prepending the collection's name.

For example:

```
<namespace>.<collection>.module
<namespace>.<collection>.role
<namespace>.<collection>.playbook
...
```

Within a corporation, it is very common having each infrastructural team creating an Ansible collection having the corporate name as namespace and the team's name as the collection name.

Ansible Galaxy

Both Ansible roles and collections can be delivered through a distribution server and managed by the `ansible-galaxy` command-line tool.

You can of course use the online Ansible Galaxy repository, or run your own, for example, using Pulp3, using it as a local caching proxy to the online one and/or as a repository where to store your own Ansible roles and collections.

Working with Ansible Roles

It has finally come the time to see an Ansible role in action.

The very first thing to do when dealing with any use case is to have a look at the online Ansible Galaxy repository to see if there is already any role for our specific use case: the online Ansible Galaxy provides a convenient web UI for quick lookups indeed – https://galaxy.ansible.com/ui/.

Install an Ansible Role

As an example, use case of an Ansible role, we will consider the use case of installing a PostgreSQL server on the `lab01.carcano.corp` target host.

As we said, instead of just starting to write a playbook, the best practice is to have a look into the online Ansible Galaxy to see if there is anything off-shelf nearly fitting our use case.

A quick check shows us that the `galaxyproject.postgresql` Ansible role exists: since it looks like an official (and so well maintained) one, using it should be safe enough also in the future to come – the goal is avoiding spending a lot of time writing a custom role and, even better, not having to maintain it.

Let's install the `galaxyproject.postgresql` Ansible role using the `ansible-galaxy` command-line utility as follows:

```
ansible-galaxy role install -p /ansible/roles \
galaxyproject.postgresql
```

The above statement runs the `ansible-galaxy` command-line tool, specifying to install the downloaded role in the `/ansible/roles` directory within the container (`-p` command-line parameter).

Write a Playbook Invoking the Role

The best way for running a role is to include it into a playbook.

Note Although it is technically possible to run a role using an Ansible one-shot statement (you must pass the role as module and specify to include roles when looking up for modules), as we saw that is a very uncomfortable way for running Ansible, suitable only for single-statement massive actions.

Some roles provide a sample playbook that is used for running unit tests beneath the `tests` subdirectory, but unfortunately that is not the case of the `galaxyproject.postgresql` Ansible role.

This means that we must guess by ourselves at how to structure our own playbook from scratch.

The `ansible/playbooks/infra/db/postgresql/deploy.yml` from the book companion resources provides a short but non-trivial example we can use as a model to elaborate.

This playbook contains a single play – let's have a closer look to it:

```
- hosts: all
  become: true
```

The play applies to every host in the inventory (`hosts: all`), running everything as the root user (`become: true`).

Then there is the following `pre_tasks` list:

```
  pre_tasks:
    - rpm_key:
        key: https://download.postgresql.org/pub/repos/yum/keys/PGDG-RPM-
        GPG-KEY{{ '-AARCH64' if ansible_facts['architecture'] == 'aarch64'
        else '' }}-RHEL
      when: "'postgresql' in host_labels and ansible_facts['os_family'] ==
      'RedHat'"
```

The `pre_tasks` list is a list of tasks run before the processing of the `roles` lists.

The above `pre_tasks` list invokes the `rpm_key` Ansible module to download the RPM GPG key used to sign the RPM packages that will be installed by the `galaxyproject.postgresql` Ansible role.

As you now know, the best practice is to verify RPM packages' GPG signature, which requires installing the RPM GPG key used to sign them.

Since the ARM RPM GPG key name contains also the `-AARCH64` token, it uses the following JINJA2 snippet to compose the correct file name:

```
{{ '-AARCH64' if ansible_facts['architecture'] == 'aarch64' else '' }}
```

This snippet returns the token `-AARCH64` only if the detected hardware architecture reported by the Ansible facts is `aarch64`. This way, on `x86_64` systems it downloads the `PGDG-RPM-GPG-KEY-RHEL` file, whereas on `aarch64` system it downloads the `PGDG-RPM-GPG-KEY-AARCH64-RHEL`.

The clause:

```
when: "'postgresql' in host_labels and ansible_facts['os_family'] ==
'RedHat'"
```

limits running the task only when the target host has the `postgresql` label and the Ansible facts report the system is part of the Red Hat OS family.

Then there is the following `roles` list:

```
roles:
  - role: galaxyproject.postgresql
    when: "'postgresql' in host_labels"
```

It instructs Ansible to run the `galaxyproject.postgresql` Ansible role only if the target host has the `postgresql` label. The aim is preventing the installation of PostgreSQL on hosts which are not supposed to have it onboard, thus avoiding human errors.

Note Using host labels in `when` conditions is a safety measure that can help to ensure tasks are run only on hosts matching specific criteria. This can be a lifesaver or at least helps to limit damages if anybody runs a playbook on an incompatible host by mistake.

Write a Var File

As you can easily guess, running the role alone is not enough: we also need to provide the necessary values to process the configuration.

There are various ways for providing values in Ansible, from adding variables to the inventory to using var files directly loaded by the playbooks or using the `-e` command-line option of the `ansible-playbook` command.

Since this is just a primer, we keep things simple: just create the `ansible/environment/postgresql-instance.yml` variable file with the following contents:

```
postgresql_version: 14
postgresql_conf:
  - listen_addresses: "'*'"
  - port: 5433
  - max_connections: 50
postgresql_pg_hba_conf:
  - host all all all md5
```

These settings instruct Ansible to deploy an instance of PostgreSQL version `14` and to configure it to listen on port `5433` on any available IP address, allowing up to `50` simultaneous connections. PostgreSQL HBA is then set to enable password-based authentication.

Note If you want to know more about these settings, please refer to the official PostgreSQL documentation.

Run a Play

We are now ready to have a go with our playbook:

```
ansible-playbook -e @environment/postgresql-instance.yml \
-l lab01.carcano.corp playbooks/infra/db/postgresql/deploy.yml
```

When prompted, type the password to unlock the public key we created for Ansible at the beginning of this chapter.

The above statement runs the `ansible-playbook` command-line tool to

- Connect to the target systems, limiting the run to the `lab01.carcano.corp` target host only
- Load the variables from the `environment/postgresql-instance.yml` var file
- Run the `playbooks/infra/db/postgresql/deploy.yml` playbook

Despite the playbook having only a few tasks, Ansible runs every task in the role that applies to the target system: depending on your system and network speed, the run can take several minutes before completing.

Once done, the last lines of the report must look like as follows:

```
PLAY RECAP *********************************************************************
lab01.carcano.corp: ok=23 changed=14 unreachable=0
failed=0    skipped=4    rescued=0    ignored=1
```

This is the play recap:

- 23 tasks were successfully run but didn't produce a change.
- 14 tasks were successfully run and produced a change.
- 4 tasks were skipped.
- 1 task was ignored.

Let's verify that the PostgreSQL service on the `lab01.carcano.corp` host is operational by running the following remote SSH statement:

```
ssh -i ansible/environment/ansible.key -l ansible \
lab01.carcano.corp "systemctl status postgresql-14.service"
```

The output must be as follows:

```
• postgresql-14.service - PostgreSQL 14 database server
     Loaded: loaded (/usr/lib/systemd/system/postgresql-14.service;
     enabled; preset: disabled)
     Active: active (running) since Fri 2025-05-30 19:43:42 UTC; 11min ago
     Docs: https://www.postgresql.org/docs/14/static/
     Process: 243284 ExecStartPre=/usr/pgsql-14/bin/postgresql-14-check-
     db-dir ${PGDATA} (code=exited, status=0/SUCCESS)
     Process: 243419 ExecReload=/bin/kill -HUP $MAINPID (code=exited,
     status=0/SUCCESS)
   Main PID: 243289 (postmaster)
      Tasks: 8 (limit: 102026)
     Memory: 16.0M
        CPU: 1.293s
     CGroup: /system.slice/postgresql-14.service
```

```
├─243289 /usr/pgsql-14/bin/postmaster -D /var/lib/
  pgsql/14/data/
├─243290 "postgres: logger "
├─243292 "postgres: checkpointer "
├─243293 "postgres: background writer "
├─243294 "postgres: walwriter "
├─243295 "postgres: autovacuum launcher "
├─243296 "postgres: stats collector "
└─243297 "postgres: logical replication launcher "
```

Ansible Vault

As you can easily guess, since values files are cleartext, they are not suitable for storing confidential information such as credentials or private keys.

To overcome this problem, the Ansible suite provides the `ansible-vault` command-line tool which generates encrypted value files that can be decrypted only by providing the decryption password.

Create a Vault File

To see it in action, let's create the `environment/postgresql-instance.vault` file as follows:

```
ansible-vault create environment/postgresql-instance.vault
```

When prompted, type the password you want to use to encrypt and decrypt this `ansible-vault` encrypted file.

Once the editor is opened, add the following contents:

```
pgsql_databases:
  - name: foo_database
    dbo_username: foo_user
    dbo_password: foo_password
  - name: bar_database
    dbo_username: bar_user
    dbo_password: bar_password
```

The above list contains dictionaries describing a few database instances: we will create them in one of the following labs, demonstrating how to use this `ansible-vault` encrypted file.

Save the changes and exit to the shell.

You can easily verify that `postgresql-instance.vault` is an encrypted file – just type:

```
sudo cat ansible/environment/postgresql-instance.vault
```

Display the Contents of a Vault File

If necessary, you can of course display the decrypted contents by running

```
ansible-vault view environment/postgresql-instance.vault
```

This obviously requires you to enter the decryption password.

Edit an Existing Vault File

Or, if you need to edit it, just run

```
ansible-vault edit environment/postgresql-instance.vault
```

Also, this operation obviously requires you to enter the decryption/encryption password.

Use a Vault File

To illustrate the use for `ansible-vault`-encrypted files, the book's companion resources provide the `ansible/playbooks/infra/db/postgresql/delivery.yml` playbook, which contains the necessary tasks to create the PostgreSQL databases we just configured in the `environment/postgresql-instance.vault` file.

Let's run it as follows:

```
ansible-playbook -J -e @environment/postgresql-instance.yml \
-e@environment/postgresql-instance.vault -l \
lab01.carcano.corp playbooks/infra/db/postgresql/delivery.yml
```

The above statement runs the `ansible-playbook` command-line tool to

- Ask for the `ansible-vault` password (`-J` option)
- Connect to the target systems, limiting the run to the `lab01.carcano.corp` target host only
- Load the variables from the `environment/postgresql-instance.yml` var file
- Load the variables from the `environment/postgresql-instance.vault` var file. Since Ansible detects it as an `ansible-vault` protected file, it automatically decrypts the contents using the provided `ansible-vault` password
- Run the `playbooks/infra/db/postgresql/delivery.yml` playbook

Note Ansible vault is just the default and most basic way for protecting sensitive information: mind anyway there are other ways of protecting confidential contents, for example, it is very easy to use Hashicorp's Vault using the `hvac` Python's module.

Work with Ansible Collections

The final section of this chapter is about how to deal with Ansible collections.

Install an Ansible Collection

As the use case for playing with a shelf Ansible collection we will run a modified version of the `join.yml` playbook we previously used, which provides the missing bits to install the `redhat-lsb-core` RPM package.

As we previously said, in order to have Ansible's `facter` to be able to detect all the available facts on the target host, on Red Hat family target hosts it is necessary to install the `redhat-lsb-core` RPM package.

The problem with this package is that on some derivatives it is provided by an optional repository which is disabled by default.

Ansible manages the enabling and disabling of DNF repositories using the `dnf_config_manager` module available in the `community.general` Ansible collection.

So, to use `dnf_config_manager` module, we must first install the `community.general` Ansible collection as follows:

```
ansible-galaxy collection install --force -p \
/ansible/collections community.general
```

The above statement runs the `ansible-galaxy` command-line tool, specifying to install the downloaded collection in the `/ansible/collections` directory within the container (`-p` command-line parameter).

In our specific setup, we also needed to specify the `--force` option: this is necessary because the Ansible version provided by the Alpine Linux container comes from the Python PyPI. Since that kind of Ansible distribution also provides a few Ansible collections, installing other ones using the `ansible-galaxy` command-line tool may break compatibility. So, in this specific use case, the force option has to be intended as "I know what I'm doing".

Invoke Modules from a Collection into a Playbook

The `ansible/playbooks/engine/join-full.yml` playbook extends the playbook `ansible/playbooks/engine/join.yml` playbook adding the following tasks:

- If the os platform is Oracle Linux, enable the `distro_builder` repository using the `dnf_config_manager` module available in the `community.general` Ansible collection.
- If the os platform is Rocky Linux, enable the `devel` repository using the `dnf_config_manager` module available in the `community.general` Ansible collection.
- Install the `redhat-lsb-core` RPM package using the `package` module available in the `ansible.builtin` Ansible collection.
- If the os platform is Oracle Linux, disable the `distro_builder` repository using the `dnf_config_manager` module available in the `community.general` Ansible collection.
- If the os platform is Rocky Linux, disable the `devel` repository using the `dnf_config_manager` module available in the `community.general` Ansible collection.

As you can see, this was also an opportunity to introduce the `when` conditional in some of the tasks to run them only on specific os platforms.

Run a Play

Let's run this playbook. First, as we already did when running the `join.yml` playbook, we must provide the password to be set to the `ansible` local system user and store it into the `ANSIBLE_SVC_PASSWORD`:

```
read -s ANSIBLE_SVC_PASSWORD
```

then, we can run the playbook:

```
ansible-playbook -u mcarcano -k \
-e targets=lab01.carcano.corp \
-e ansible_svc_user=ansible \
-e ansible_svc_password=${ANSIBLE_SVC_PASSWORD} \
playbooks/engine/join-full.yml
```

The run must be completed successfully, installing also the `redhat-lsb-core` package.

Again, remember to unset the `ANSIBLE_SVC_PASSWORD` variable:

```
unset ANSIBLE_SVC_PASSWORD
```

Structuring the Playbooks Directory Tree

Before closing the chapter and finishing this long (yet very essential) book, it is worth giving a few pieces of advice about how to structure the playbooks directory tree.

Ansible automations should be part of a shared project, developed and managed by multiple individuals.

Requiring uniform behavior without governance and guidelines is unrealistic. Therefore, it is best practice to standardize the structure and criteria for organizing playbooks to prevent the project from becoming unmanageable.

A clean, tidy, and predictable structure allows team members to quickly identify available resources and determine what needs to be developed. This approach promotes

reusability and improves maintainability by reducing duplication and preventing code divergence. Additionally, fixes are automatically inherited by all playbooks that import shared resources, enhancing overall reliability.

Ultimately, these practices lead to streamlined processes that effectively address complex use cases while avoiding maintainability issues.

Note The best practice is to write a Solution Definition Document for Ansible itself, which includes a lookup table to manage this structure. Being Agile compliant is about enabling people to deliver solutions quickly and iteratively, not forcing them to improvise everything because there are no standards and governance. That is another framework called chaos.

In my experience, a good structure consists of creating two distinct directory trees:

- `infra`, used to group playbooks for infrastructure deployment and configuration management. These playbooks are generally developed, maintained, and operated by infrastructural specialists such as system engineers, database administrators, network engineers, and so on.
- `solutions`, dedicated to the playbook for deploying and managing business-level solutions services consisting of one or more technological services and their dependencies. These playbooks are generally developed, maintained, and operated by application delivery teams, since they are specialized in specific solutions.

To help you better visualizing this structure, since in our workshop the `ansible/playbooks/infra` directory tree already exists, let's create the missing `ansible/playbooks/solutions` as follows:

```
mkdir ansible/playbooks/solutions
```

The Infrastructural Playbooks Directory Tree

The labs presented in this chapter are clearly infrastructural use cases. We indeed put PostgreSQL-related playbooks into the `infra` directory tree, beneath the `db/postgresql` subdirectory.

It contains

- A playbook dedicated to deploying and configuring PostgreSQL (`deploy.yml`)
- A playbook dedicated to deploying database instances (`delivery.yml`)

As you can see, beneath the `infra` directory, there is the `db` directory, representing the technology family, which contains the `postgresql` directory, describing the actual technology managed by the playbook.

To illustrate the concept with a general rule, my advice is to structure the `infra` directory with these two nesting levels:

- Family, such as `linux`, `windows`, or `db`
- Technology (`mysql`, `postgresql`, `mssql`, ...) or components (`firewalld`, `firewall`, ...)

Examples are

- `infra/linux/firewalld`
- `infra/windows/firewall`
- `infra/db/mysql`
- `infra/db/postgresql`
- `infra/www/apache`
- `infra/www/nginx`
- `...`

Note Depending on your automation codebase size and needs, you may also want to consider creating a leaf directory for versioning, such as `v1`, `v2`, and so on.

To improve knowledge sharing and promote onboarding, each callable playbook must have in the same directory a related documentation file explaining how to invoke it, the available tags, and so on.

For example:

- `deploy.yml` must have `deploy.md`
- `delivery.yml` must have `delivery.md`

And so on.

The Solutions Playbooks Directory Tree

Solutions delivery is a more complex topic, since it must also consider the delivery of the solution's requirement.

Managing requirements can greatly benefit from having a standard `infra` director tree, importing the necessary task list from the `infra` playbooks as necessary.

As for the solution-specific playbook, the best practice in my experience is to create beneath the `solutions` directory a subdirectory dedicated to the specific solution.

Mind that the name must not be related to a specific instance deployment: it must just refer to the solution.

For example, assuming the solution to deploy is Taiga (an open source project management software similar to JIRA), we create the `solutions/taiga` directory containing playbooks such as `deploy.yml` (manage the deployment process, including the upgrade), `projects.yml` (operate on Taiga projects, such as creating new ones and modifying or deleting the existing ones), and so on. The `deploy.yml` playbook will just import the requisite step lists for the related technologies, such as `postgresql`, from the `infra` directory tree.

To improve knowledge sharing and promote onboarding, each callable playbook must have in the same directory a related documentation file explaining how to invoke it, the available tags, and so on.

For example:

- `deploy.yml` must have `deploy.md`
- `projects.yml` must have `projects.md`

And so on.

The last part consists of providing instance-specific blueprints with the proper settings: these blueprints can simply be stored beneath the `environment/blueprints` directory, using a subdirectory for each available environment.

For example:

- `environment/blueprints/lab`
- `environment/blueprints/dev`
- `environment/blueprints/test`
- `environment/blueprints/prod`

As for the manifests, they should have a name reflecting the instance, such as:

- `taiga-p1-0.yml` Taiga Production Security Tier 1 Instance 0
- `taiga-l1-0.yml` Taiga Lab Security Tier 1 Instance 0

and so on.

The blueprint is then passed to the `ansible-playbook` command line as a vars file, to run the solution-specific playbook with the proper values for the specific instance.

This structure provides several benefits: the most remarkable are a nearly declarative approach provided using blueprints and consistency across instances granted by using the same playbooks when operating.

Summary

This chapter gave you a big picture of what Ansible is and how to exploit it to run powerful automations.

More specifically we learned

- How to set up a containerised Ansible on a workstation
- How to configure the environment so that it can be run from the shell without explicitly invoking Docker, providing a very friendly user experience in the same way, enabling it to run statements as it was really installed on the system
- How to deal with confidential settings using Ansible vault
- Writing a playbook for preparing hosts for being managed by Ansible
- How to use Ansible Galaxy for downloading and installing shelf Ansible roles and collections
- How to instantiate roles from specific collections

- A few guidelines for structuring the directory tree containing all the automations playbooks and manifest files

For DevOps and DevSecOps professionals, understanding how to install and operate Ansible running playbooks and roles in a standardized manner is a critical skill, since it enables running automation processes ensuring they are repeatable and reliable, reducing the risk of errors during deployment. Also knowing how to operate Ansible in ad hoc mode is essential, especially for troubleshooting and immediate task execution.

Utilizing Ansible Vault is particularly important for maintaining security in DevSecOps practices. It enables professionals to securely manage sensitive data, such as passwords and API keys, within playbooks. This ensures that security best practices are followed, protecting critical information from unauthorized access.

Working in a standardized way with Ansible is crucial for DevOps and DevSecOps professionals for several reasons. First, a well-defined structure for organizing playbooks and roles enhances collaboration among team members, allowing for efficient knowledge sharing and onboarding. By adhering to a consistent directory structure, professionals can quickly locate resources, understand dependencies, and maintain clarity in complex automation projects.

In summary, mastering Ansible's standardized practices, including ad hoc operations, playbook execution, and secure data management, empowers DevOps and DevSecOps professionals to deliver high-quality, secure, and efficient automation solutions that align with organizational goals.

Conclusion

Here it ends this book: I hope you enjoyed it and it helped you understand DevOps and DevSecOps on Linux in practice.

In this long writing, before delving into technical concepts, we discovered together the importance of teamwork to reach goals. We then worked on improving your holistic vision by having an overview of the necessary methodologies such as Agile and Lean and frameworks such as SCRUM and Kanban.

We then started the technical part from the basics of development creating a good baseline with the BASH shell and how to use it for scripting. Since they are very frequently used in both DevOps and DevSecOps scopes, we also defined a good baseline with regular expressions.

We then started the hands-on part of the book, which is used as a companion to the theoretical concepts throughout the book: we played with several engaging shell scripts, example statements with the most used command-line tools, and developed a full-featured Python project, including its integration in a CI/CD.

The initial hands-on are shell scripting examples along with statements with the most used tools, drawn up to cover the most common needs in DevOps and DevSecOps scopes.

After introducing basic data formats (like key-value and INI) and how to process them, we delved into standard formats, like XML, JSON, and YAML, which are a mandatory part of the skill set of DevOps and DevSecOps professionals. To see these concepts in action, we learned how to process them not only using standard tools such as xpath, xmlstarlet, jq, and yq, but also using Python standard libraries.

We then focused on creating a solid system security baseline: starting from basic concepts such as POSIX permissions, POSIX ACLs, and mount point security options, we then moved to more advanced practices such as configuring and using sudo, assigning Linux capabilities, and SELinux,

As for software security, we learned the importance of running static code scan analysis: focusing on Python, we learned how to enforce corporate development standards by using compliance check scanners and how to use vulnerability scanners to ensure software security. This includes not only scanning for vulnerabilities in the developed source code but also the imported third-party software libraries.

In the context of modern software development, we learned software best practices, such as promoting early error detection by creating and running unit tests using the PyTest framework.

Since data confidentiality both on disk and on the wire is mandatory nowadays, with the support of hands-on labs, we learned how to implement confidentiality using Gnu GPG, OpenSSL, x509 certificates, Public Key Infrastructures, and Transport Layer Security (TLS).

Since security best practices strongly recommend always using a centralized authentication source, we also delved into the LDAP protocol and how to implement corporate-wide authentication using LDAP.

From the system integration perspective, we delved into must-know protocols such as HTTP, REST, and SOAP, which are the most used industrial-level protocols nowadays. We also learned in practice how to exploit Gatehill's Imposter to mock real REST and SOAP services to provide convenient targets for running component tests.

From the software engineering perspective, we learned best practices focusing on Python: we saw how to structure an application in a self-contained directory tree, how to turn it into a full-featured Python project, including how to generate wheel packages and RPM packages. We then learned how to use Makefiles to automate the application's installation in both self-contained directory trees beneath the user's home directory and the standard Linux system path and to automate the generation of Python wheel and even RPM packages.

We delved into Linux container technologies, starting from a deep insight of Docker and Podman, along with a good overview of Kubernetes and Helm. The concepts were supported by several engaging hands-on labs which can be run on Killercoda. The example labs also gave us the opportunity to play with Pulp3, which is a very handy brick to use in a CI/CD Toolchain.

We also illustrated how to install Rancher on a single Docker container image, providing a powerful Kubernetes-based platform you can keep after completing the book and exploit for your own Labs. To provide a non-trivial use case, we deployed Gitea and a Gitea Runner using the official Helm chart.

We then introduced the fundamentals of continuous integration, continuous delivery, and continuous deployment, highlighting components and teams' roles. We also explained the importance of Change Management in Continuous Deployment. To see these concepts in action, we also deployed a Gitea Action workflow implementing the software release process by running the unit tests and then packaging a Python application as an RPM, completing the process by publishing it into a Pulp3 repository.

The last part of the book provides a thorough explanation of Git, including several hands-on labs illustrating the explained concepts.

Lastly, after a propaedeutic introduction to SSH, we learned how to properly work with Ansible, illustrating with hands-on examples how to set up a full-featured container-based management station with inventory and Ansible-vault-protected secrets. Using that management station, we learned hands-on how to write playbooks, including implementing ones embedding third-party roles.

I hope you enjoyed this book: I designed it balancing theory and hands-on labs trying to provide you with the necessary concepts to become a valuable professional yet keeping the flow engaging, tailoring examples and labs to the real world, providing enough theory to let you fully understand the many practical labs inside it.

Some chapters, such as the ones dealing with Agile, Lean, SCRUM, Kanban, and CI/CD, contain a blended summary of standards, technologies, and frameworks you are supposed to thoroughly know. For the sake of brevity, I limited myself to a blended summary, providing a reference to the original works.

To continue your training, if you did not already, I strongly encourage you to read the following books:

- **For ITIL**: *ITIL Foundation, ITIL 4 Edition* by AXELOS (available from AXELOS or authorized publishers)
- **For the PDCA cycle**: *Out of the Crisis* by W. Edwards Deming
- **For DMAIC and Six Sigma**: *The Six Sigma Way* by Peter S. Pande, Robert P. Neuman, and Roland R. Cavanagh or official Six Sigma resources from organizations like ASQ (American Society for Quality)
- **For Agile**: "Manifesto for Agile Software Development" (freely available at agilemanifesto.org under a Creative Commons license)
- **For Lean**: *Lean Software Development: An Agile Toolkit* by Mary Poppendieck and Tom Poppendieck
- **For SCRUM**: *The Scrum Guide* by Ken Schwaber and Jeff Sutherland (freely available at scrum.org under a Creative Commons license)
- **For Kanban**: *Kanban: Successful Evolutionary Change for Your Technology Business* by David J. Anderson or *Kanban from the Inside* by Mike Burrows
- **For introductory DevOps and CI/CD concepts**: *DevOps for Dummies* by Emily Freeman and Joel Bancroft-Connors
- **For CI/CD and toolchain concepts**: *The DevOps Handbook* by Gene Kim, Jez Humble, Patrick Debois, and John Willis
- **For continuous delivery practices**: *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* by Jez Humble and David Farley

Index

A

Abstract syntax tree (AST), 322
Access Control Lists (ACLs), 374
Ad hoc mode, 790
Agile
  Agile Manifesto, 27
  framework, 30, 31
  project management methodology, 28
  scope, 31, 32
    macro, 32, 33
    project managers, 34
  step-by-step approach, 29
Amazon Elastic Container Registry (ECR), 491
American Society for Quality (ASQ), 12
Ansible
  ad hoc statements, 781, 790–793
  collections, 800, 808–810
  container image, 782
  galaxy, 800
  namespaces, 798
  playbook, 793–797
  playbooks directory tree, 810–814
  roles, 799
    installation, 801
    playbook invoking, 801–803
    run play, 804, 805
    write a Var file, 803
  setup container environment
    configuration, 784–786
    directory tree, 783
    launching shell, 788, 789
    pull, 783
    run, 786–788
  vault, 806–808, 815
ANT or Maven, 288
Apache HTTPd, 357
Apache httpd container image, 518
API gateway, 448
Apple MAC, 774
Arrays, 76
Associative array, 99
Asymmetric cryptography, 372
Automated Certificate Management Environment (ACME) protocol, 418
AWK
  built-in variables, 174–182
  CI/CD pipelines, 159
  control statements, 169, 170
  fields variables, 161
  gods-of-it.csv file, 160
  logical operators, 167
  match() built-in function, 168
  pattern-action statement, 163
  patterns, 164–166
  real-life examples, 182–185
  scripts, 159
  shell invocation, 161, 162
  variables, 171, 172

B

Backlog grooming, 40
Bandit, 322

© Marco Antonio Carcano 2026
M. A. Carcano, *DevSecOps and DevOps for Linux*, https://doi.org/10.1007/979-8-8688-2077-9

BASH cheat sheet
control structures, 89–91
flow control, 87–89
functions, 92–94
list and sequences, 83, 84
numbers, 79, 80
script positional arguments, 74
strings, 81–83
testing file and directories, 86
trapping signals, 95
user input, 84, 85
variables, 75
arrays, 76, 77
empty/non-empty/length, 78
operations, 76
special, 77
Basic regular expressions (BRE), 115, 116, 125
Behavior-Driven Development (BDD), 269
BitKeeper, 688
Borg, 554
Bottom-up approach, 34
Bourne Again SHell (BASH), 49, 50
Branches
checkout and managing, 713–715
merging, 728
remote, 721–726, 728
stashing, 716–721
Branch merging strategies
Carcano organization, 729
fast-forward, 731, 732
ORT, 737–740, 742
rebase/fast-forward, 733–737
resolve conflicts, 747–749
revert merge, 745, 746
squashed merge, 742, 743, 745
statements, 729
buildah command-line tool, 521

C

Certificate Authority (CA), 415, 428, 583
Certificate Revocation Lists (CRLs), 408, 423, 426, 428, 436
Certificate Signing Request (CSR), 410, 418
Chaos, 811
Charts, 571
CI/CD toolchain
automated testing facilities, 646
build automation, 645
components, 641
repository manager, 642–644
SCM, 639, 641
software development lifecycle, 655
software development process, 640
Cloud services, 641
Command-line interface (CLI), 495
Command-line utilities, 49
Comma-separated values (CSV), 106
Common Vulnerabilities and Exposures (CVEs), 296
Concurrent Versions System (CVS), 688
Configuration Management Database (CMDB), 11
Containerfile, 521
Container management suites
Docker, 494–496
Podman, 496, 497
Container object, 469
Containers
features, 486
filesystem, 487
image registries, 489–491
images, 487–489
Container Storage Interface (CSI), 570
Continuous deployment (CD)

customer-facing repository, 652, 653
essential checks, 655
processes, 650
staging environment, 651, 652
strategies example, 653, 654
Continuous integration (CI)
continuous building pipeline, 648
continuous testing pipeline, 649
development, 647, 648
processes, 647
Continuous integration and deployment (CI/CD), xxxv, 269
CRL Distribution Points (CDP), 423, 428
Crypto-refresh, 379
curl directory, 699
Custom-defined tags, 253
Custom Resource Definitions (CRDs), 569

D

Data Leak Prevention (DLP), 449
Debian-based systems, 295
Definitive Media Library (DML), 11
Dictionaries, 99, 250
diff command-line utility, 701
Discretionary Access Control (DAC), 360
Docker, xxxvii, 493–496, 521, 556
docker-compose.yml, 526
Dockerfile, 521
Docker Hub, 490
Document Type Definitions (DTD), 189, 217
Don't Repeat Yourself (DRY) approach, 92, 192, 256, 264, 799

E

egrep, 130
Encryption algorithms, 761
/etc/yum.repos.d directory, 402
eval command, 768
Extended regular expressions (ERE), 115, 125
Extensible Markup Language (XML)
attributes, 192
configuration files, 189, 194
definition, 190
DTD file, 198–200
fancymonitor, 214–216
hands-on approach, 193
principles, 190
schema, 195–197
schema languages, 190
tag types, 191
XPath, 202–207, 209
XSD file, 200, 201
XSLT, 210–213
Extensible Stylesheet Language (XSL), 210
Extensible Stylesheet Language Transformations (XSLT), 210
Extreme programming (XP), 29, 269

F

Fancymonitor, 262, 278, 285
f-docs, 717
Feature-driven development (FDD), 29
Field-matching pattern, 164
Flake8, 316–319
Fully Qualified Domain Names (FQDNs), 407, 458, 504, 760

G

Gatehill's Imposter, 537, 539
getfattr and setfattr, 343
__getattribute__ method, 237

Git, 439
  areas, 689–690
  branches, 690, 691
  code reviews, 687
  commands, 688
  commits, 310
  commits compliance, 310
  configuration, 309, 692, 693
  creating repositories, 693–696
  hooks, 754, 755
  installation, 691
  pre-commit hook
    configuration, 311, 312
    installation, 311
    reset branch, 313
    upgrading scanners, 312, 313
  remote repositories, 710–712
  tags, 691
  transport protocols, 696
  version operations
    checking differences, 700, 701
    commit, 703, 704
    committing contents, 703
    comparing commits, 708, 710
    listing changes, 704, 705
    reverting specific commit, 705, 706
    staging modified contents, 699, 700
    unstaging contents, 701, 702
    working area, 698
git diff command, 700
Gitea, 491, 694
  action runners, 615–618
  actions workflows, 599, 621, 622, 638
  charts, 608
  CI/CD solution, 599
  create organizations, 619
  deployment, 614
  Helm charts, 606–610
    action runners, 629
    configure domain, 623
    configure migration sources, 624
    configure resource requests and limits, 634
    configure webhooks, 624
    enable monitoring, 633
    LDAP authentication, 631, 632
    LoadBalancer, 625, 626
    logging configuration, 630, 631
    scaling, 635, 636
    TLS certificate, 626–628
    values, 610–613
    version, 625
  repositories, create, 620, 621
  requisites, 601, 603–606
  testing instance, 600
  web UI login, 614
Gitea Actions Runners, 613, 658
Gitea actions workflows, 683
  create Fancymonitor repository
    add remote origin, 661, 662
    organizations, 659, 660
    pushing main branch to remote origin, 662
    unit test workflows, 663–666
  creating release, 675–679, 681
  mocking services, 657, 666–670
  pull request, 671
    completion, 674, 675
    merging strategies, 671
    pushing branch, 672
    raising request, 672, 673
Gitea Cloud, 600
GitHub, 526, 694
GitHub Actions, 658
Gitlab, 491, 694
GitOps, 654

git reset command, 706
Git Version Control System (VCS), 309
Global Regular Expressions Print (Grep), *see* grep
GNU Make
  build automation tool, 287, 288
  definition, 288
  Fancymonitor project, 287
  installation, 289
  Makefile, 289
  real-life example, 290
GNU Privacy Guard (GnuPG/GPG), 374, 379
  best practices, 404
  encryption and decryption contents, 389
  Linux ecosystem, 377
  operations, 384–388
  primary keys, 381, 382
  RPM packages, 378
  signed RPM, 400–403
  signing applications
    detached applications signature, 394
    package-specific embedded signature, 395–399
  signing contents
    clear-text signature, 391
    detached signature, 392, 393
    signed document, 390
    usage scenarios, 390
  subordinate keys, 383
  WoT, 380
Go-Live, 23
Google Container Registry (GCR), 491
gpgkey directive, 402
grep
  automated scripts, 132
  case-insensitive scan, 129
  definition, 127–128
  invert the matching criteria, 129
  pipe, 130
  scanning every file, directory tree, 129
  scanning files, 128
  shell scripts, 131
  validating standard input, 130
Gzipped tarballs, 277

H

HEAD, 715
Helm, 553
  add repositories, 571
  charts contents, 572, 573
  CPU and RAM, 570
  deploy charts, 573–575
  installation, 571
  search charts, 572
Heredoc, 62, 63
Here-string (<<<), 62
Huawei's Software Repository for Container (SWR), 491
Hypertext Transfer Protocol (HTTP)
  clear-text application-layer protocol, 439
  proxies
    outgoing, 449, 450
    reverse, 448, 449
  request messages, 440–442
  response messages, 444–447
  virtual hosts, 447

I

Image vulnerability scanning, 491
Infrastructure as code (IaC), 241
Ingress Rule, 566

J

Java KeyStore (JKS), 430
JavaScript Object Notation (JSON)
  arrays, 231–234
  complex use cases, 234–238
  configuration files, 219
  editing with VIM, 220, 221
  JQ, 221–224, 239
  serialization
    comments, 225
    complex data types, 229, 230
    null value, 225
    scalars, 226–228
  syntax, 219
JFrog's Artifactory, 491

K

Kanban, 815
  to add, 42
  meetings, 44
  roles, 42
Keep It Simple, Stupid (KISS) approach, 6
Keystore/Keystore Containers, 430
Key-value store (KVS), 99
  CSV, 106, 108–111
  directory, 100
  INI, 102–105
  script processes, 101
K3s, xxxvii, 555, 578
kubeadm, 554
kubectl command-line tool, 556
Kubernetes, 553
  CRDs, 569, 570
  developer community and
    enterprises, 554
  kubeadm, 554
  kubectl command-line tool, 556–558
  namespaces, 556
  resource definitions
    ConfigMap, 568
    deployment, 560
    Ingress Rule, 566, 567
    pod, 559
    PV, 563, 564
    PVC, 563
    ReplicaSet, 561
    secret, 568, 569
    service, 564–566
    StatefulSet, 562

L

Laboratory (Lab), 11
ldapsearch, 480
ldapsearch command-line utility, 479
Lean, 27
  methodologies, 41
  principles, 30
Lightweight Directory Access
    Protocol (LDAP)
  application protocol, 465
  Base DN, 471
  bind DN, 476
  command line tools, 479
  enterprise-class application, 465
  hierarchical structure, 468–470
  instantiating objects, 471–473
  ldapsearch command-line
    client, 476–479
  OpenLDAP, 467
  referrals, 475, 476
  schema and objectclasses, 467
  search filters, 473–475
Line-matching pattern, 164
LinkedIn, 4

Linux, 335
Linux distributions, 296
Linux filesystem, 278
lsscsi command, 161

M

Makefiles, 288
Mandatory Access Control (MAC) model, 333, 360
Master keys, 258, 381
Meta-characters, 116
Microservices, 533
Microsoft Active Directory, 466
Microsoft Windows, 485
Minikube, 554
Mocking services, 666
Mutual TLS (mTLS) authentication, 463

N

Network File System (NFS), 367
Network Information Service (NIS), 466
NoSQL databases, 100

O

OAuth2, 324
Object Identifier (OID), 467
Online Certificate Status Protocol (OCSP), 408, 423, 426, 436
OpenAPI, 544–546, 549
Open Container Initiative (OCI) standards, 494–496
OpenLDAP, 467
OpenPGP, 379
OpenSCAP, 196
Openshift, 555
OpenSSL, 333
Operating container and images
  cleanup, 524
  creation, 521
  networking and volumes, 522, 523
  Podman, 498
  registries
    authentication, 500–502
    local container, 498
    searching container images, 504–508
    spin up local container, 498, 499
    unqualified search registries, 504
    untrusted registries, 503
  running containers
    checkpoint and restore, 514–517
    operation, 509, 512
Oracle Linux 9, xxxvi
Ostensibly Recursive Twin (ORT), 737
OverlayFS, 487

P

Parsed Character Data (PCDATA), 199
-p command-line option, 775
Perl-Compatible Regular Expressions (PCRE), 115, 125
Persistent Volume (PV), 563, 587
Persistent Volume Claims (PVCs), 563, 570, 587, 601
Phony targets, 290
Plan-Do-Check-Act (PDCA) cycle, 11, 16
Playbooks, 793
Pod, 559
Podman, 493, 496, 497
Portable Operating System Interface (POSIX), 333, 336
POSIX ACLs (Access Control Lists), 333

Pretty Good Privacy (PGP), 379
printf command, 80
Private keys, 387
Product backlog, 38
ProxyJump directive, 771
Pruning, 708
Public Key Cryptography Standards (PKCS), 408
Public Key Infrastructure (PKI), 407, 415, 426
Pulp, 526
Pulp3, docker compose
  CI/CD, 525, 526
  create repositories, 531
  deployment, 527, 528
  install Pulp CLI, 531
  login to webUI, 530
  YAML file, 533
Pylint, 317–319
PyPI repository, 644
PyTest, 270
Python, 279
Python Package Index (PyPI), 244
Python Packaging Authority's (PyPa), 279
PyYAML library, 241

Q

Quai.io, 490

R

Rancher, xxxvii, 555
  public clouds or on-premises, 576
  single-node container
    active directory
      authentication, 592–595
    backups, 587–592
    CA, 583–586
    components, 577
    deployment, 580–582
    operate K3s, 582
    requirements, 578–580
Rancher charts, 576
Red Hat, 555
Regular expressions (RegEx)
  capturing group/backreferences, 123, 124
  class of characters, 117, 118
  logical OR, 122
  meta-characters, 116, 117
  negating class of characters, 122
  purpose, 115, 116
  quantification, 119–122
  shorthand, 118–119
  text parsing and manipulation, 115
Relative Distinguished Name (RDN), 469
ReplicaSet, 561
Representational State Transfer (REST), 239, 543
Rockylinux 9, 518
Role-based access control (RBAC), 491, 556
RPM (for Red Hat-based systems)
  build packages, 298, 299
    build Fancymonitor, 302, 303
    spec file, 300–301
  operation, 304–307
  package managers, 296
  packaging format, 296, 297
rpmbuild command, 298
RUN command, 519

S

Scalars, 226
Scientific Linux, 296

Scripts
execution programming, 67
shebang, 68, 70
special LC_ALL=C locale, 70
static analysis tools, 73
time utility, profiling, 73
writing comments, 71, 72
Scrum
ceremonies, 39–41
fast-paced environments, 46
frameworks, 35
project, 38
roles, 36
sprints, 39
time-boxed methodology, 36
SCRUM, 815
Secondary keys, 383
Section or stanza, 102
Secure Shell (SSH)
authentication
password-based, 764
public key, 764–768, 770
client configuration files, 770, 771
connecting to server, 772, 773
host key fingerprint, 762, 763
host keys, 761, 762
primary materials, 759
running remote commands, 773, 774
SCP, 774, 775
security issues, 761
server security practices, 776–778
SFTP, 776
Security-Enhanced Linux (SELinux), 333, 360, 374
AVC violations, 363
file content management, 368, 369
module booleans, 367
modules, 366
operational status, 361–363
policy sets, 360
tags, 363–365
TE rules, 365, 366
Security Information and Event Management (SIEM) software, 123
Server Name Indication (SNI) extension, 457, 463
Shell
built-in functions, 50
command-line interface, 50
command-line utilities, 51–55
commands and pipelines, 56
Heredoc, 62–64
here-string, 62
input and output redirection, 59
options, 55
pipeline option, 58
streams and file descriptors, 59–61
subshells, 57
Xargs, 65, 66
Simple Certificate Enrollment Protocol (SCEP), 418
Simple Object Access Protocol (SOAP), 189
benefits, 538
components, 538
mocking endpoint, 539–543
mocking REST endpoint, 546–548
REST API, 543
WSDL, 539
Skopeo, 524
Software Defined Networks (SDN), 486
Software delivered as a service (SaaS), 653
Software packaging
CI/CD, 277
Fancymonitor application, 277, 278
Python, 279
Python build module, 280, 281

Software packaging (*cont.*)
setuptools, 279
source distribution, 284
tests, 282, 283
wheel distribution, 284
Solution lifecycle
continuous improvement process, 25
functional and non-functional requirements, 26
operating process, 24
service design documents, 20
service design process, 16–19
service transition process, 21, 24
Sonatype's Nexus, 491
Source Code Management (SCM) software, 641, 642, 694
Source Control Management (SCM), 11
SPEC file, 298
Sprints, 39
ssh-agent, 779
StatefulSet, 563
Static code analysis tools
code quality, 315
Flake8, 316, 317
Pylint, 317–319
Step-by-step approach, 29
Stream EDitor (sed)
backreferences, 156
command-line tool, 133
commands, 146, 147, 153–156
expression syntax, 145, 146
hold space, 144, 145
in-place file modification, 136, 137
invocation, 134, 135
line matching clauses, 149–153
line-oriented design, 157
multiple expressions, 148
nesting expressions, 147
pattern space, 140–143
record matching clauses, 138
showing changes, 139
Subject Alternative Names (SANs), 412, 418, 457
sudo command, 350
Swiss NTP pool, 136
Symmetric cryptography, 369
Systems security
cryptography
asymmetric, 372–374
symmetric, 369–372
directories, 334
files and directories
date and time attributes, 336
extended attributes, 343–345
POSIX, 336–340
POSIX ACLs, 345–347
POSIX special permissions, 340, 343
Linux capabilities, 356–359
mount options, 349, 350
SELinux, 360
sudo, 350–355

T

Tags, 251
checking out, 752
creation, 751
delete, 752
listing, 751
remote, 752–754
repository-wide scope, 750
tarball, 54
TCP port 80 traffic, 451
TCP port 443 traffic, 451
Teamwork
achievements, 5

co-workers, 9
incident reports, 6
internal problems, 3
self-confident, 4
service request, 6
tasks and requests, 8
unnecessary meetings, 7
unneeded communications, 8
Test-Driven Development (TDD), 269
tomcat directory, 260
Transport Layer Security (TLS), 420
available policies, 454
clear-text protocols, 453
computer network, 453
handshake, 455
mutual TLS authentication, 462
plain-text protocols, 453
Red Hat family distributions, 454
server certificate bundle, 460, 461
server certificate validation, 458, 459
SNI, 457
Type Enforcement (TE) rules, 360, 366

U

Unit tests
continuous testing and collaboration, 276
JUnit, 276
PyTest framework, 270
installation, 271
real-life cycle, 271–275
use, 269, 270
Untrusted registries, 503

V

Vagrantfile, xxxviii
Version Control Systems (VCSs), 641, 688
Virtualbox, xxxviii
Virtual File System (VFS), 335
Virtual machines (VMs), xxxv, xxxvii, 485, 760
Vulnerability scanning tools
Bandit, 322, 323
non-compliance, 321
safety, 324–329

W

Web Application Filtering (WAF), 448
Web of trust (WoT), 380
Web Services Description Language (WSDL), 189, 539

X

X.509 certificates, 374
application-specific truststores, 434–436
CA
ACME protocol, 419–423
CSR, 418
generate private key, 417
PKI, 415, 416
submit CSR, 418
types, 415
keystores, 430–432
private CA, 433
self-signed certificate, 408–414
SSL/TLS, 407
validation, 423–429
XML Path Language (XPath), 189, 202
XML Schema Definitions (XSD), 189, 217, 538
XML Schema Document (XSD), 200
XSLT stylesheet, 210

Y, Z

YAML
advantage, 242
serialization, 246
anchors and aliases, 256–258
comments, 247
dictionaries, 250
document separator, 247
indentation-based and flow-style notations, 246
lists, 251
merge key, 258, 259
node types, 247
null value, 248
scalars, 248, 249
tags, 251, 253
versions and schemas, 254, 255
yamllint, 259, 260
VIM, editing, 242, 243
YAML, Python, 260–264
Yq, 244, 245
yamllint command-line utility, 259
yamlloader function, 263
Yellowdog Updater, Modified (YUM), 402
yum command-line utility, 643