



Red Hat Training and Certification

Student Workbook (ROLE)

OCP 4.6 DO400

Red Hat DevOps Pipelines and Processes: CI/CD with Jenkins, Git, and Test-driven Development (TDD)

Edition 5



Red Hat
Learning Community

Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



Network with tens of thousands of community members



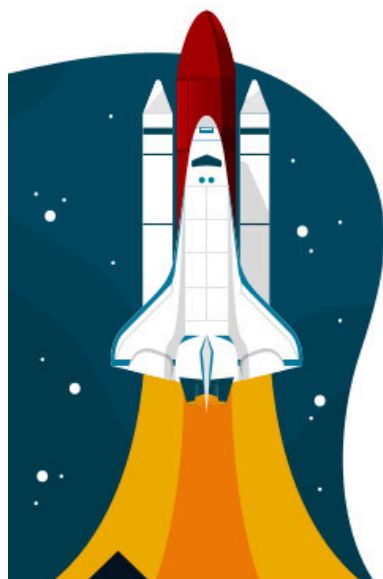
Engage in thousands of active conversations and posts



Join and interact with hundreds of certified training instructors



Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

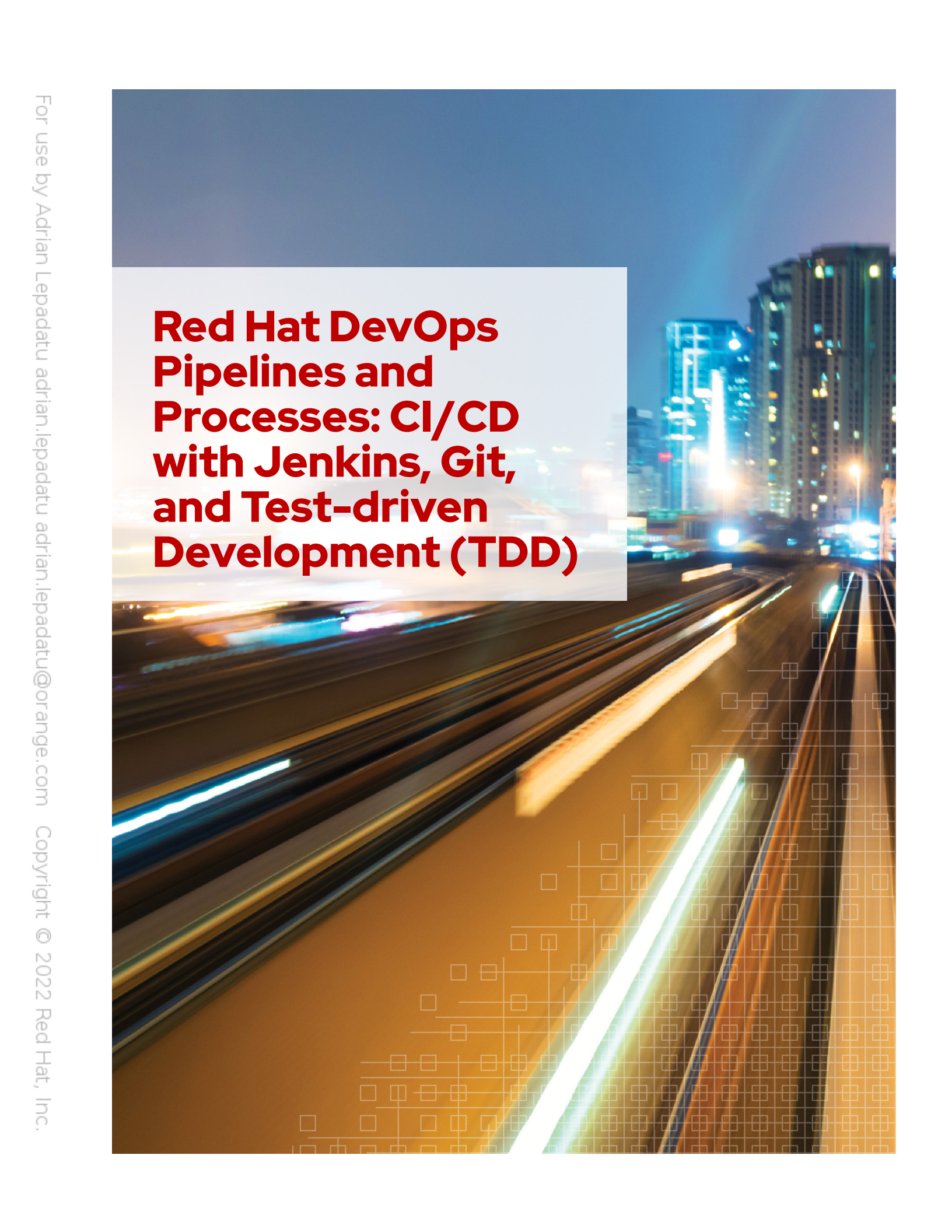
Connect with your instructor – and your classmates – before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

The background of the slide is a long-exposure photograph of a city street at night. The image shows multiple lanes of traffic with light trails from cars and streetlights, creating a sense of motion. In the background, there are tall buildings with lit windows. A semi-transparent white rectangular box is overlaid on the left side of the image, containing the title text. The bottom right corner of the slide features a faint, light-colored grid pattern.

Red Hat DevOps Pipelines and Processes: CI/CD with Jenkins, Git, and Test-driven Development (TDD)

OCP 4.6 DO400

Red Hat DevOps Pipelines and Processes: CI/CD with Jenkins, Git, and Test-driven Development (TDD)

Edition 5 20211123

Publication date 20211123

Authors: Aykut Bulgu, Enol Álvarez de Prado, Guy Bianco IV,
Eduardo Ramirez Ronco, Jaime Ramírez Castillo, Jordi Sola Alaball,
Pablo Solar Vilariño
Editor: Sam Ffrench

Copyright © 2021 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are Copyright © 2021 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send email to training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, OpenShift, Fedora, Hibernate, Ansible, CloudForms, RHCA, RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a registered trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation or the OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: David Sacco, Richard Allred, Zachary Gutterman

Document Conventions	ix
.....	ix
Introduction	xi
Red Hat DevOps Pipelines and Processes: CI/CD with Jenkins, Git, and Test-driven Development (TDD)	xi
Orientation to the Classroom Environment	xii
1. Introducing Continuous Integration and Continuous Deployment	1
Defining CI/CD and DevOps	2
Quiz: Defining CI/CD and DevOps	8
Describing Jenkins and Pipelines	10
Quiz: Describing Jenkins and Pipelines	14
Guided Exercise: Configuring a Developer Environment	18
Summary	33
2. Integrating Source Code with Version Control	35
Introducing Version Control	36
Quiz: Introducing Version Control	39
Building Applications with Git	41
Guided Exercise: Building Applications with Git	45
Creating Git Branches	52
Guided Exercise: Creating Git Branches	59
Managing Remote Repositories	67
Guided Exercise: Managing Remote Repositories	79
Releasing Code	90
Guided Exercise: Releasing Code	97
Lab: Integrating Source Code with Version Control	107
Summary	117
3. Implementing Unit, Integration, and Functional Testing for Applications	119
Describing the Testing Pyramid	120
Quiz: Describing the Testing Pyramid	126
Creating Unit Tests and Mock Services	130
Guided Exercise: Creating Unit Tests and Mock Services	138
Creating Integration Tests	151
Guided Exercise: Creating Integration Tests	156
Building Functional Tests	164
Guided Exercise: Building Functional Tests	171
Lab: Implementing Unit, Integration, and Functional Testing for Applications	176
Summary	187
4. Building Applications with Test-driven Development	189
Introducing Test-driven Development	190
Guided Exercise: Introducing Test-driven Development	193
Developing with TDD and Other Best Practices	206
Quiz: Developing with TDD and Other Best Practices	211
Analyzing Code Quality	219
Guided Exercise: Analyzing Code Quality	227
Lab: Building Applications with Test-driven Development	239
Summary	252
5. Authoring Pipelines	253
Building a Basic Declarative Pipeline	254
Guided Exercise: Building a Basic Declarative Pipeline	262
Creating Scripted Pipelines	271
Guided Exercise: Creating Scripted Pipelines	275
Controlling Step Execution	283

Guided Exercise: Controlling Step Execution	293
Lab: Authoring Pipelines	309
Summary	320
6. Deploying Applications with Pipelines	321
Building Images and Deploying to OpenShift	322
Guided Exercise: Building Images and Deploying to OpenShift	326
Deploying Applications to Various Environments	338
Guided Exercise: Deploying Applications to Various Environments	343
Implementing Release Strategies	357
Guided Exercise: Implementing Release Strategies	363
Executing Automated Tests in Pipelines	370
Guided Exercise: Executing Automated Tests in Pipelines	375
Lab: Deploying Applications with Pipelines	381
Summary	403
7. Implementing Pipeline Security and Monitoring Performance	405
Implementing the Basic Principles of DevSecOps	406
Guided Exercise: Implementing the Basic Principles of DevSecOps	410
Implementing Container Security Scans	420
Guided Exercise: Implementing Container Security Scans	425
Defining Performance Metrics	433
Guided Exercise: Defining Performance Metrics	437
Configuring Error Notifications and Alerts	449
Guided Exercise: Configuring Error Notifications and Alerts	452
Recovering Failures	462
Guided Exercise: Recovering Failures	466
Lab: Implementing Pipeline Security and Monitoring	479
Summary	498
A. Creating a Quay Account	499
Creating a Quay Account	500

Document Conventions

This section describes various conventions and practices used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:



References

These describe where to find external documentation relevant to a subject.



Note

These are tips, shortcuts, or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on something that makes your life easier.



Important

These provide details of information that is easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring these admonitions will not cause data loss, but may cause irritation and frustration.



Warning

These should not be ignored. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Red Hat DevOps Pipelines and Processes: CI/CD with Jenkins, Git, and Test-driven Development (TDD)

DevOps practices have enabled organizations to undergo a digital transformation; moving from a monolithic waterfall approach, to a rapidly deploying cloud-based agile process. This transformation requires a team of developers trained to use tools that enable them to spend more time coding and testing and less time troubleshooting.

DevOps Pipelines and Processes: CI/CD with Jenkins, Git, and Test-driven Development (TDD) is a 4-day introduction to DevOps for developers that teaches students the necessary skills and technologies for automated building and deploying of cloud-native applications.

Course Objectives

- Build essential skills to implement Agile and DevOps development processes and workflows.

Audience

- Microservice Application Developers.
- Platform Developers.

Prerequisites

- Experience with application development in Java, Node.js, Python, or others is required.
- The course *OpenShift or Introduction to OpenShift Applications (DO101)* is strongly recommended, but not required.
- The course *Red Hat Application Development I: Programming in Java EE (AD183)* is recommended, but not required.
- Be proficient in using an IDE such as Red Hat® Developer Studio or VSCode.

Orientation to the Classroom Environment

DO400 is a **Bring Your Developer Workstation (BYDW)** class, where you use your own internet-enabled system to access the shared OpenShift cluster. The following operating systems are supported:

- Red Hat Enterprise Linux 8 or Fedora Workstation 32 or later
- Ubuntu 20.04 LTS or later
- Microsoft Windows 10
- macOS 10.15 or later

BYDW System Requirements

<i>Attribute</i>	<i>Minimum Requirements</i>	<i>Recommended</i>
CPU	1.6 GHz or faster processor	Multi-core i7 or equivalent
Memory	8 GB	16 GB or more
Disk	10 GB free space HD	10 GB or more free space SSD
Display Resolution	1024x768	1920x1080 or greater

You must have permissions to install additional software on your system. Some hands-on learning activities in DO400 provide instructions to install the following programs:

- Python 3
- Node.js
- JDK
- Git 2.18 or later (Git Bash for Windows systems)
- The OpenShift CLI (oc) 4.6.0 or later

You might already have these tools installed. If you do not, then wait until the day you start this course to ensure a consistent course experience.



Important

Hands-on activities also require that you have a personal account on GitHub, a public, free internet service.

BYDW Systems Support Considerations

Depending on your system, you might see differences between your command-line shell and the examples given in this course.

Red Hat Enterprise Linux or Fedora Workstation

- If you use Bash as the default shell, then your prompt might match the `[user@host ~]$` prompt used in the course examples, although different Bash configurations can produce different results.
- If you use another shell, such as `zsh`, then your prompt format will differ from the prompt used in the course examples.
- When performing the exercises, interpret the `[user@host ~]$` prompt used in the course as a representation of your system prompt.
- All the commands from the exercises should be functional.

Ubuntu

- You might find differences in the prompt format.
- In Ubuntu, your prompt might be similar to `user@host:~$`.
- When performing the exercises, interpret the `[user@host ~]$` prompt used in the course as a representation of your Ubuntu prompt.
- All the commands from the exercises should be functional.

macOS

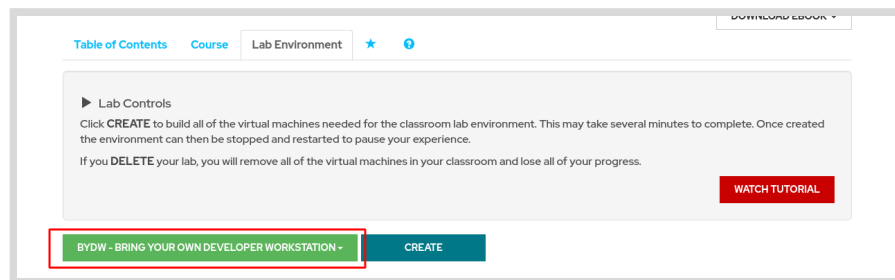
- You might find differences in the prompt format.
- In macOS, your prompt might be similar to `host:~ user$`.
- When performing the exercises, interpret the `[user@host ~]$` prompt used in the course as a representation of your macOS prompt.
- All the commands from the exercises should be functional.
- You might need to grant execution permissions to the installed runtimes.

Microsoft Windows

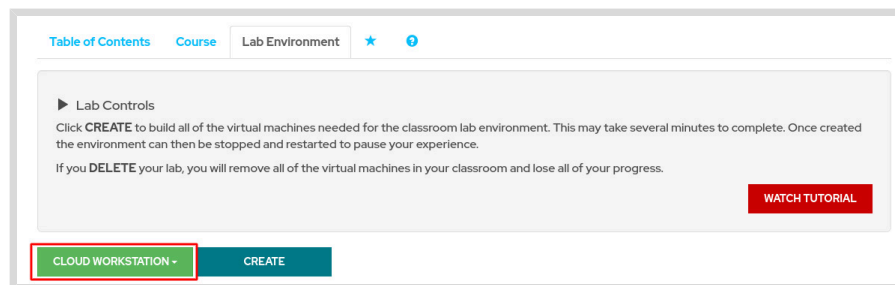
- Windows does not support Bash natively. Instead, you must use PowerShell.
- In Windows PowerShell, your prompt should be similar to `PS C:\Users\user>`.
- When performing the exercises, interpret the `[user@host ~]$` Bash prompt as a representation of your Windows PowerShell prompt.
- For some commands, Bash syntax and PowerShell syntax are similar, such as `cd` or `ls`. You can also use the slash character (`/`) in file system paths.
- For other commands, the course provides help to transform Bash commands into equivalent PowerShell commands.
- This course only provides support for Windows PowerShell.
- The Windows firewall might ask for additional permissions in certain exercises.

Creating a Lab Environment

To use your own system or an ILT workstation provided by your instructor, select the **BYDW** option from the dropdown when you launch your classroom by using the **CREATE** button in the **Lab Environment** tab in the Red Hat Online Learning (ROL) interface.



If you cannot or do not wish to use your own system, ROL can also provide a RHEL 8 workstation environment in the cloud, which you can connect to remotely from your browser. To use this, select the **Cloud Workstation** option from the dropdown when you launch your classroom by using the **CREATE** button in the **Lab Environment** tab in the ROL interface.



For all classrooms provisioned for this course, Red Hat Online Learning (ROL) also provisions an account for you on a shared Red Hat OpenShift 4 cluster. When you provision your environment in the ROL interface, the system provides the cluster information. The interface gives you the OpenShift web console URL, your user name, and your password.

OpenShift Details		
Username	RHT_OCP4_DEV_USER	your-user
Password	RHT_OCP4_DEV_PASSWORD	yourpassword
API Endpoint	RHT_OCP4_MASTER_API	https://api.CLUSTER.prod.nextcle.com:6443
Console Web Application		https://console-openshift-console.apps.CLUSTER.prod.nextcle.com
Cluster Id		37bd2501-4358-41b9-9d1b-56946f7f765

The required tools are pre-installed in the **Cloud Workstation** classroom environment, which also includes VSCode, a text editor that includes useful development features.

Cloud Workstation Classroom Overview



Important

The remaining information in this section explains the **Cloud workstation** or non-BYDW ILT classroom environments and is not relevant to you if you are using the BYDW option with your own system.

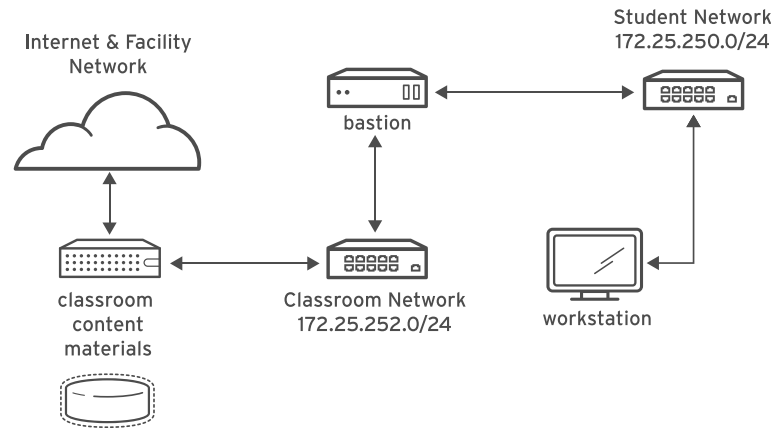


Figure 0.4: Cloud workstation classroom overview

In this environment, the main computer system used for hands-on learning activities is **workstation**. All virtual machines in the classroom environment are in the `lab.example.com` DNS domain.

All student computer systems have a standard user account, `student`, which has the password `student`. The root password on all student systems is `redhat`.

Classroom Machines

Machine Name	IP Addresses	Role
<code>workstation.lab.example.com</code>	172.25.250.9	Graphical workstation used by students
<code>bastion.lab.example.com</code>	172.25.250.254	Router linking student's VMs to classroom servers
<code>classroom.lab.example.com</code>	172.25.252.254	Server hosting the classroom materials required by the course

The **bastion** system acts as a router between the network that connects the student machines and the classroom network. If **bastion** is down, other student machines may not function properly or may even hang during boot.

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning classroom. They are accessed through a web application hosted at `rol.redhat.com` [<http://rol.redhat.com>]. You should log in to this site using your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through a web page. The state of each virtual machine in the classroom is displayed on the page under the **Online Lab** tab.

Machine States

Virtual Machine State	Description
STARTING	The virtual machine is in the process of booting.
STARTED	The virtual machine is running and available (or, when booting, soon will be).
STOPPING	The virtual machine is in the process of shutting down.
STOPPED	The virtual machine is completely shut down. Upon starting, the virtual machine boots into the same state as when it was shut down (the disk will have been preserved).
PUBLISHING	The initial creation of the virtual machine is being performed.
WAITING_TO_START	The virtual machine is waiting for other virtual machines to start.

Depending on the state of a machine, a selection of the following actions is available.

Classroom/Machine Actions

Button or Action	Description
PROVISION LAB	Create the ROL classroom. Creates all of the virtual machines needed for the classroom and starts them. Can take several minutes to complete.
DELETE LAB	Delete the ROL classroom. Destroys all virtual machines in the classroom. Caution: Any work generated on the disks is lost.
START LAB	Start all virtual machines in the classroom.
SHUTDOWN LAB	Stop all virtual machines in the classroom.
OPEN CONSOLE	Open a new tab in the browser and connect to the console of the virtual machine. You can log in directly to the virtual machine and run commands. In most cases, you should log in to the workstation virtual machine and use ssh to connect to the other virtual machines.
ACTION → Start	Start (power on) the virtual machine.
ACTION → Shutdown	Gracefully shut down the virtual machine, preserving the contents of its disk.
ACTION → Power Off	Forcefully shut down the virtual machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION → Reset	Forcefully shut down the virtual machine and reset the disk to its initial state. Caution: Any work generated on the disk is lost.

At the start of an exercise, if instructed to reset a single virtual machine node, click **ACTION → Reset** for only the specific virtual machine.

At the start of an exercise, if instructed to reset all virtual machines, click **ACTION** → **Reset**

If you want to return the classroom environment to its original state at the start of the course, you can click **DELETE LAB** to remove the entire classroom environment. After the lab has been deleted, you can click **PROVISION LAB** to provision a new set of classroom systems.



Warning

The **DELETE LAB** operation cannot be undone. Any work you have completed in the classroom environment up to that point will be lost.

The Autostop Timer

The Red Hat Online Learning enrollment entitles you to a certain amount of computer time. To help conserve allotted computer time, the ROL classroom has an associated countdown timer, which shuts down the classroom environment when the timer expires.

To adjust the timer, click **MODIFY** to display the **New Autostop Time** dialog box. Set the number of hours until the classroom should automatically stop. Note that there is a maximum time of ten hours. Click **ADJUST TIME** to apply this change to the timer settings.

Chapter 1

Introducing Continuous Integration and Continuous Deployment

Goal

Describe the principles of DevOps and the role of Jenkins.

Objectives

- Define Continuous Integration, Continuous Delivery, and Continuous Deployment.
- Describe Pipelines and the features of Jenkins.

Sections

- Defining CI/CD and DevOps (and Quiz)
- Describing Jenkins and Pipelines (and Quiz)
- Configuring a Developer Environment (Guided Exercise)

Defining CI/CD and DevOps

Objectives

After completing this section, you should be able to define Continuous Integration, Continuous Delivery, and Continuous Deployment.

Explaining DevOps

In the Information Technology (IT) industry, a key factor for success is the ability to deliver applications quickly and reliably. Software and systems development today are evolving from monolithic operated projects, to distributed, microservices-based applications. These new architectures require a degree of automation and team cohesion, which traditional software development approaches do not meet.

DevOps is an approach to software and systems engineering intended to speed up the application delivery process, maximize quality, and increase business value. Based on the culture of collaboration, automation, and innovation, DevOps aims to guide the full development and delivery process, from the initial idea to the production deployment.

Main Characteristics of DevOps

DevOps is often described as a culture, a mindset, or a movement. The term is a result of the combination of *development* and *operations*, which have been traditionally separate IT departments. This might suggest that DevOps is just a blend of both departments, but its implications go further than just merging developers and operators.

The DevOps culture aims to guide the full development software life cycle, promoting changes across different levels of an organization, such as business processes, people collaboration, delivery practices, and quality assurance. To this end, DevOps focuses on the following:

Culture of collaboration

The main idea behind the DevOps term is to eliminate the barrier between the development and operations teams. This simple idea promotes open communication and emphasizes the performance of the entire system.

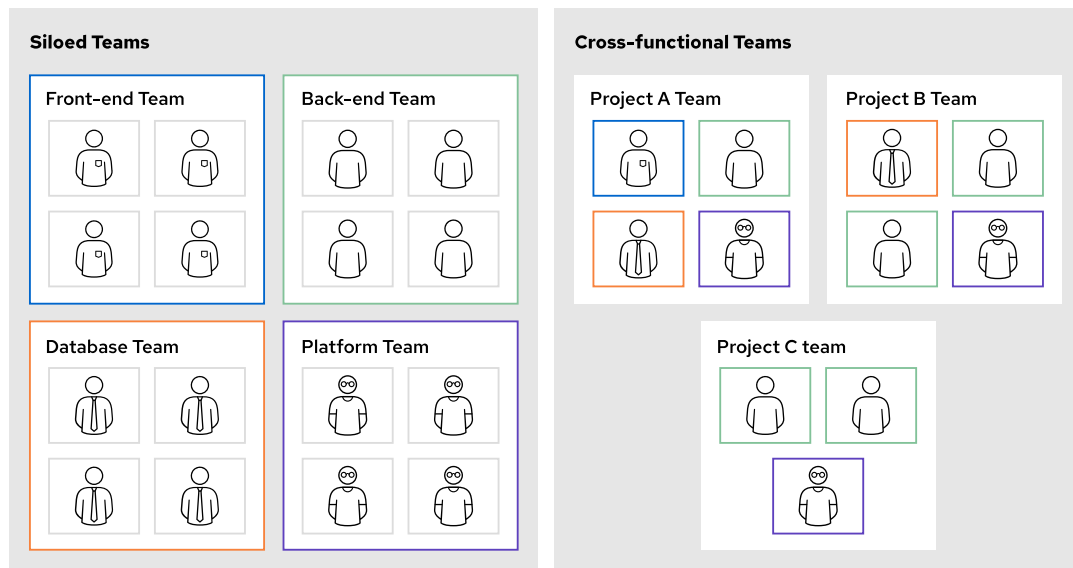


Figure 1.1: Cross-functional teams break down knowledge silos

Knowledge silos disappear, in favor of cross-functional autonomous teams, which have fewer dependencies to bring their work to production. Examples of good collaboration practices are **pair programming** and **code reviews**.



Note

Code review, peer review, or peer code review is the practice of reviewing the code of a program to assure quality. Because the cost of fixing a bug in production is higher than the cost in earlier stages, developers usually review code before integrating changes. Code reviews are also an effective way for developers to discuss changes and share knowledge.

Pair programming is a development discipline in which two developers work in collaboration, usually in the same workstation. With pair programming, programmers share knowledge, collaborate to solve problems, and help each other by reviewing code.

Maximizing automation

To achieve a frequent, responsive, and reliable process, you must fully incorporate automation as a key practice in your development and delivery processes. Prioritize automation wherever automation is viable and avoid manual and repetitive tasks, which consume time, produce errors, and undermine the morale of the team. A highly automated process also improves the responsiveness of the delivery process, with shorter feedback loops and increased quality.

Automation is at the core of the DevOps mindset. To maximize the automation of development and delivery tasks, you must push towards the use of key techniques, such as **Continuous Testing, Continuous Integration, Continuous Delivery, and Continuous Deployment**. You will learn about these techniques and core concepts throughout the course.

Continuous Improvement

DevOps is often defined as a *journey* of continuous experimentation, with failures as drivers for improvement. DevOps fosters continual experimentation and understands that iteration and practice are the prerequisites to mastery.

Never stop trying to improve your processes and pay special attention to your automated tests. A clean and complete set of tests will encourage you to experiment further and take more risks.

Describing Continuous Integration

Software developers working on a specific feature normally do so in isolation from the main line of development. Classic ways of isolating work in progress is to work in a local copy and use version-control branches. For example, teams commonly use the Git version control system and a principal branch called *main* to integrate their changes into.

After developers finish a feature, they integrate the feature branch into the main branch. If a branch remains isolated from the main branch for a long time, then the integration process becomes more difficult and prone to errors.

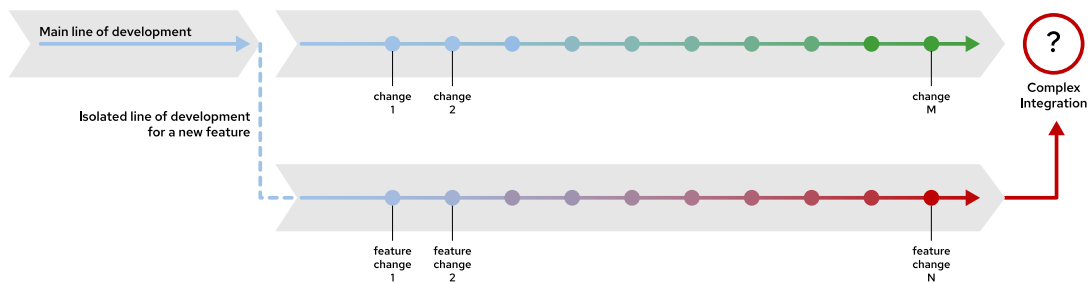


Figure 1.2: Long-lived isolated branches are difficult to integrate into the main branch

Continuous Integration (CI) is the discipline of **integrating changes in the main branch as often as possible**. Developers use short-lived branches or small change sets and integrate them frequently into the main branch, ideally several times a day. This speeds up the integration, makes code review easier, and reduces potential problems.

Continuous Integration normally entails validation of integrated changes by using automated integration tests. After the changes have been integrated and validated, the team can then decide when to deploy a new release. By itself, Continuous Integration does not involve automating the release process.

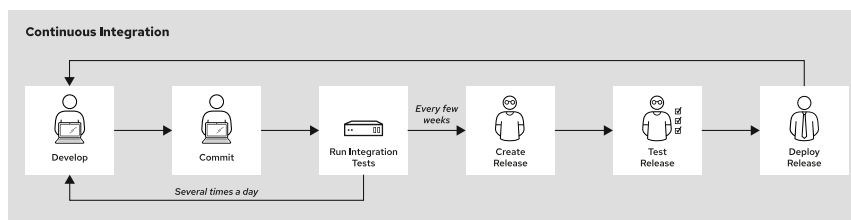


Figure 1.3: Continuous Integration

Implementing CI is an excellent way to begin your DevOps transformation. After introducing CI in your process, you will have the foundations necessary to adopt Continuous Delivery and Continuous Deployment.

Continuous Integration Tools

At a high level, you need the following tools to implement Continuous Integration:

- **A version control system**, which stores your source code in a repository. The repository uses a *main* or *trunk* branch to keep track of the main development line. Feature development occurs

in separate branches, which after review and validation, are integrated into the main branch. The most popular version control system is Git.

- **A CI automation service.** This is usually a server or a daemon that watches the repository for changes. If the repository changes, the CI automation service checks out the new code and verifies that everything is correct. Popular tools used for Continuous Integration are Jenkins, Tekton, and GitLab.

Verifying Changes Automatically with Continuous Integration

When the repository changes, the CI service runs an automated build to verify that the new changes are correct. This build is usually a series of verification stages, which, as you will learn later in the course, form what is called a **CI pipeline**. A basic CI pipeline is usually comprised of the following stages:

1. **Checkout:** the CI service detects changes in the repository and downloads the new changes.
2. **Build:** the CI service executes the commands necessary to build and package the application.
3. **Test:** the CI service runs the application tests to verify that the latest changes are correct.

If any of these stages fail, then the CI server should mark the current state of the project as broken. A broken pipeline impacts or blocks the development flow.

Continuous Integration in DevOps

Although Continuous Integration existed before DevOps, the new DevOps culture has adopted Continuous Integration as one of its core practices. With a DevOps mindset, Continuous Integration takes advantage of additional automation and collaboration techniques, such as the following:

- **Code reviews.** Changes are reviewed by peers before the code is integrated into the main branch. Tools, such as GitHub and GitLab, offer web UIs to encourage code reviews.
- **Pipelines as code.** CI pipelines are declared as source files included in the application repository. The CI configuration is therefore version-controlled and available to anyone who has access to the repository.
- **Continuous testing.** When test suites are integrated into the Continuous Integration process, they offer early and quick feedback about the quality of the changes. Additionally, continuously running tests keeps them in an active state, and prevent tests from becoming abandoned.
- **CI/CD.** In a DevOps scenario, Continuous Integration pipelines increase the automation level with Continuous Delivery and Continuous Deployment. These techniques automate the creation of releases and deployments. The *CI/CD* term refers to the combination of Continuous Integration and Continuous Delivery or Deployment.

Describing Continuous Delivery

Continuous Delivery automates the creation of application releases, further reducing manual, error prone processes. After the verification steps, the CI pipeline typically includes release creation, a deployment step, or both. The pipeline creates a deployable release every time the integration pipeline runs successfully. Ideally, this process happens several times a day.

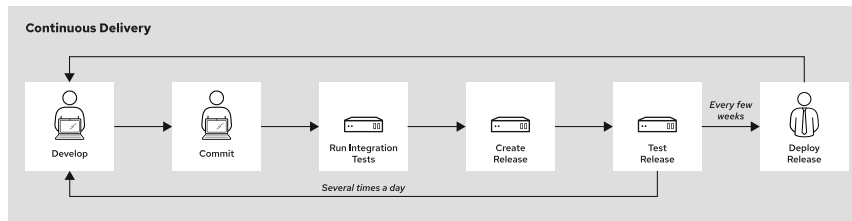


Figure 1.4: Continuous Delivery

As changes are integrated into the main branch, new releases are created automatically. The release deployment step, however, still **requires human intervention**. When teams decide to deploy a new release, they pick the release version they want to deploy and trigger the release process manually.

Not all created releases are deployed. Usually, deployments occur at certain hours, days, or weeks set by the team, but usually not several times a day.

Describing Continuous Deployment

Continuous Deployment takes the automation level even further. Release deployment is triggered automatically, **without human intervention**, usually after the rest of the validation steps have successfully completed.

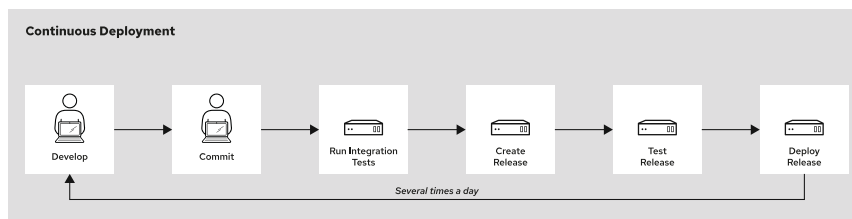


Figure 1.5: Continuous Deployment

Just like Continuous Delivery, the pipeline creates a deployable release after each successful run. For every change integrated into the main branch, the CI server generates and deploys a new release. This means that deployments occur several times a day.

You need a high level of automation and a solid testing strategy to adopt Continuous Deployment. Combined with techniques such as Test Driven Development (TDD), robust and complete automated tests ensure quality and makes the team confident enough to automate deployments without human intervention. With deployments happening several times a day, the delivery process becomes more responsive and the feedback loops shorter.

Complex CI/CD pipelines often combine both deployment approaches. For example, a pipeline can use Continuous Deployment for development and staging environments, and Continuous Delivery for the production environment.

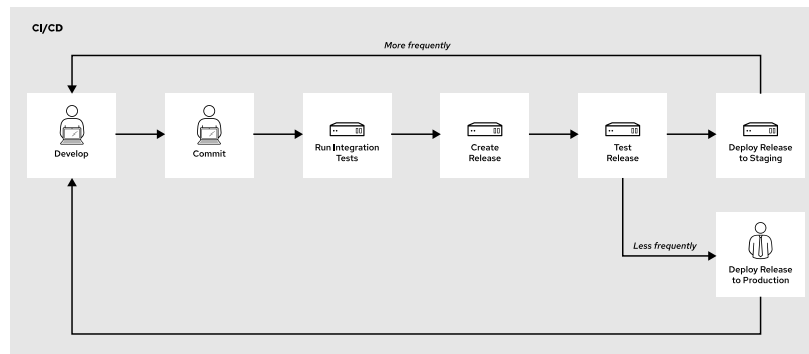


Figure 1.6: Example of CI/CD: Continuous Deployment for the staging environment. Continuous Delivery for the production environment



References

Understanding DevOps

<https://www.redhat.com/en/topics/devops>

A Survey of DevOps: Concepts and Challenges

<https://doi.org/10.1145/3359981>

The Three Ways: The Principles Underpinning DevOps

<https://itrevolution.com/the-three-ways-principles-underpinning-devops/>

Continuous integration

https://en.wikipedia.org/wiki/Continuous_integration

Continuous Integration

<https://openpracticelibrary.com/practice/continuous-integration/>

Continuous Delivery

<https://openpracticelibrary.com/practice/continuous-delivery/>

Continuous Deployment

<https://openpracticelibrary.com/practice/continuous-deployment/>

Continuous Integration

<https://martinfowler.com/articles/continuousIntegration.html>

What is CI/CD?

<https://opensource.com/article/18/8/what-cicd>

► Quiz

Defining CI/CD and DevOps

Choose the correct answers to the following questions:

- 1. **Which three stages are normally part of a Continuous Integration pipeline? (Choose three.)**
 - a. Checkout
 - b. Build
 - c. Refactoring
 - d. Testing
 - e. Debugging
- 2. **How does the Continuous Delivery strategy trigger deployments?**
 - a. Without human intervention
 - b. With human intervention
 - c. Both with and without human intervention
 - d. The Continuous Delivery strategy does not trigger deployments
- 3. **You want to adopt Continuous Deployment in your CI/CD pipeline to fully automate the build, test and deployment of your application. Which sequence of stages is required to accomplish this?**
 - a. Code checkout, build, test, release creation, and deployment with human intervention
 - b. Build, code checkout, test, and deployment with human intervention
 - c. Code checkout, build, test, release creation, and deployment without human intervention
 - d. Code checkout, build, test, and release creation
- 4. **Which three of the following practices are part of the DevOps culture? (Choose three.)**
 - a. CI/CD
 - b. Manual testing
 - c. Automation
 - d. Improved collaboration and communication
 - e. Manual deployment
- 5. **What does DevOps aim to improve for the software development life cycle?**
 - a. Reliable development and delivery process
 - b. Improved deployment frequency
 - c. Increased business value
 - d. Quality maximization
 - e. All of the above

► Solution

Defining CI/CD and DevOps

Choose the correct answers to the following questions:

- 1. **Which three stages are normally part of a Continuous Integration pipeline? (Choose three.)**
 - a. Checkout
 - b. Build
 - c. Refactoring
 - d. Testing
 - e. Debugging
- 2. **How does the Continuous Delivery strategy trigger deployments?**
 - a. Without human intervention
 - b. With human intervention
 - c. Both with and without human intervention
 - d. The Continuous Delivery strategy does not trigger deployments
- 3. **You want to adopt Continuous Deployment in your CI/CD pipeline to fully automate the build, test and deployment of your application. Which sequence of stages is required to accomplish this?**
 - a. Code checkout, build, test, release creation, and deployment with human intervention
 - b. Build, code checkout, test, and deployment with human intervention
 - c. Code checkout, build, test, release creation, and deployment without human intervention
 - d. Code checkout, build, test, and release creation
- 4. **Which three of the following practices are part of the DevOps culture? (Choose three.)**
 - a. CI/CD
 - b. Manual testing
 - c. Automation
 - d. Improved collaboration and communication
 - e. Manual deployment
- 5. **What does DevOps aim to improve for the software development life cycle?**
 - a. Reliable development and delivery process
 - b. Improved deployment frequency
 - c. Increased business value
 - d. Quality maximization
 - e. All of the above

Describing Jenkins and Pipelines

Objectives

After completing this section, you should be able to describe Pipelines and the features of Jenkins.

Describing Pipelines

The DevOps principles rely strongly on automation. To achieve reliability, cohesion, and scalability, the DevOps approach makes use of automation tools to implement pipelines.

A *pipeline* is a series of connected steps executed in sequence to accomplish a task. Usually, when one of the steps fails, the next steps in the pipeline do not run. In software development the main goal of a pipeline is to deliver a new version of software. Subsets of steps that share a common objective constitute a pipeline *stage*. Common pipeline stages in software development include the following:

Checkout

Gets the application code from a code repository.

Build

Combines the application source with its dependencies into an executable artifact.

Test

Runs automated tests to validate that the application matches the expected requirements.

Release

Delivers the application artifacts to a repository.

Validation and Compliance

Validates the quality and security of the application artifacts.

Deploy

Deploys the application artifacts to an environment.

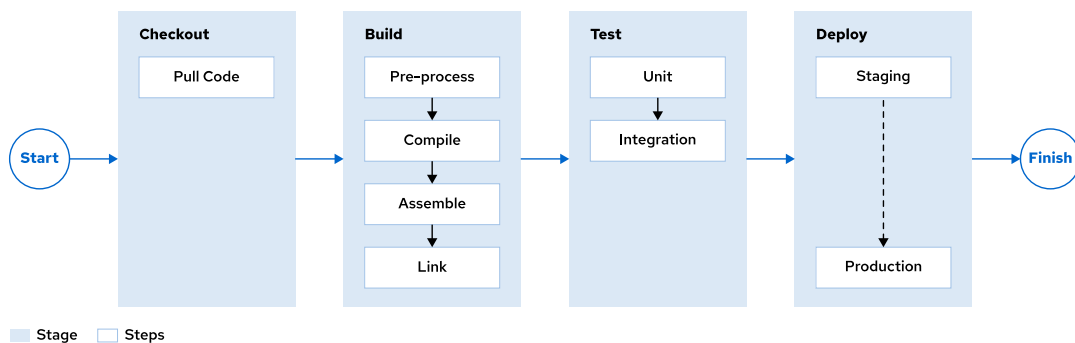


Figure 1.7: Execution flow example of a pipeline

The preceding diagram depicts the execution flow of a pipeline. The pipeline flow is flexible. You can have different stages and steps depending on your project requirements.

Defining the Pros and Cons of Using Pipelines

The use of pipelines has both advantages and drawbacks. The main advantage of using pipelines is the reduction of manual and repetitive tasks. Other advantages of using pipelines for software development are the following:

- Improves release cycles and the time-to-market of your application
- Generates more effective feedback by understanding the status of each stage in a pipeline run
- Improves code quality by adding automated testing stages
- Adds transparency and accountability for changes made to the code and to the pipeline configuration

The primary disadvantage of using pipelines is that it inherently increases the complexity of the project by adding another system, which requires monitoring and maintenance. This drawback is often vastly outweighed by the benefits that automation and pipelines provide.

Defining Pipeline Best Practices

You can define your pipelines in a way that fits your development and delivery process. You can also use best practices when defining these pipelines. The following are some of the best practices for designing and implementing pipelines:

Pipeline as code

You store the pipeline configuration in a version control system next to the application or the infrastructure code. For example, you can use Git to store both your application code and the pipeline definition file. The main advantage of using pipelines as code is that you can keep track of pipeline changes. This allows you to perform code reviews, collaborate on the development, and rollback changes to the pipeline.

Design parallel workflows

A pipeline workflow is composed of stages executed in order. This does not mean that the pipeline needs to always follow a linear flow. In some cases, you can execute stages in parallel to speed up the pipeline execution. For example, running different test suites in parallel can greatly improve pipeline performance.

Build artifacts once

A single pipeline execution should create artifacts once and reuse the artifacts in the rest of the pipeline stages. For example, a pipeline that builds a Java application would first create a *JAR* executable artifact and then use that artifact to subsequently create a container image, release, and deployment.

This best practice is related to the **ReLease** stage, in which the pipeline pushes the software artifacts to a repository for later use. Building artifacts once also allows you to optimize time and resource consumption.

Verify on a production-like environment

Create an environment identical to production to test and validate any changes before the changes are pushed to the production environment. Testing in a production-like environment reduces configuration mismatches between the development and the production environments. This strategy helps teams to detect production-specific errors earlier in the pipeline.

For example, you can create a staging environment that is as similar to production as possible. You can then configure your pipeline to deploy to this staging environment to check that everything works fine before deploying to production.

Fail fast

Stop the pipeline execution when a stage fails, this allows you to quickly address issues. A *fail fast* mentality is necessary if you want to shorten your feedback loops, which is one of the goals of the DevOps mindset.

Placing a compilation step as an early stage in the pipeline is an example of this practice. If the application does not compile, then the pipeline will fail fast, without having to execute the rest of the stages.

Describing Jenkins

To implement a pipeline, you need a tool that manages and automates the execution of all of the stages. Jenkins is one of the most popular open source tools that helps automate building, testing, and deploying applications.

Jenkins is an automation engine that supports multiple automation patterns, such as pipelines, scripted tasks, and scheduled jobs. This course covers the pipeline automation pattern.

Jenkins is written in Java, which makes it available for multiple operating systems. You can install Jenkins in macOS, Windows, and popular Linux distributions.

The Jenkins core has limited functionality, but extends its capabilities with the use of plug-ins. You can find plug-ins to integrate your pipeline with version control systems, container runtimes, and cloud providers, among others.

**Note**

Use Plugins Index [<https://plugins.jenkins.io/>] to find Jenkins plugins.

Defining the Jenkins Architecture

The Jenkins architecture was designed for distributed build environments. This architecture allows you to use different environments for each build.

A Jenkins environment consists of two logical components: a *controller* and an *agent*.

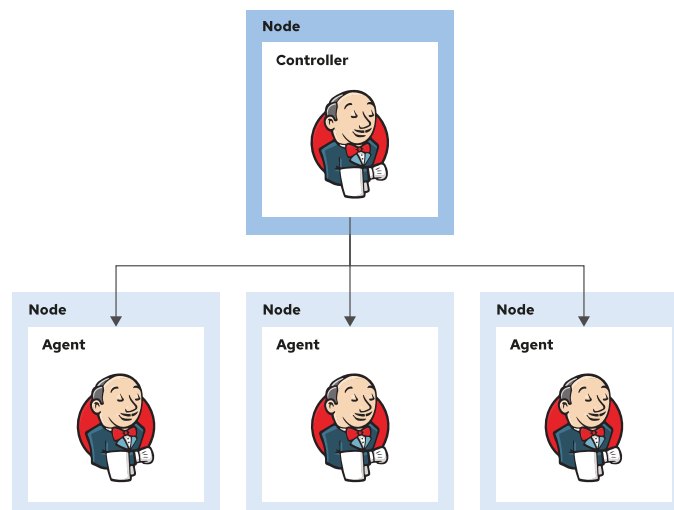


Figure 1.8: Jenkins architecture

A *controller* is a coordinating process that administers the Jenkins agents and orchestrates their work. The main responsibilities of controllers are the following:

- Stores the configuration
- Loads plug-ins
- Schedules project executions
- Dispatches projects to the agents for execution
- Monitors the agents
- Records and presents the build results

An *agent* is a machine, or a container, connected to a Jenkins controller that executes the tasks sent by the controller.

A *node* is a machine capable of executing pipelines or projects. The controllers and agents are considered nodes.

In stand-alone mode, the Jenkins controller is also capable of executing projects.

Building Jenkins Pipelines

The Pipeline plug-in extends the core functionalities and allows you to implement and integrate CI/CD pipelines in Jenkins. On pipeline-enabled projects, you must define the workflow as text scripts by using a Pipeline Domain-Specific Language (DSL). The definition of a pipeline can be written by using two types of syntax:

Declarative

A relatively new feature that is designed to make writing and reading pipeline code easier. This syntax is closer to a declarative programming model.

Scripted

Syntax that follows in a limited form of Groovy syntax. This syntax is closer to an imperative programming model.

Both types of syntax are Domain-specific Languages and are based on the Groovy programming language. You will learn how to use both the declarative and the script syntax later in the course.



References

What is a CI/CD pipeline?

<https://www.redhat.com/en/topics/devops/what-cicd-pipeline>

The Continuous Delivery Foundation

<https://cd.foundation>

Jenkins project

<https://www.jenkins.io>

Jenkins glossary

<https://www.jenkins.io/doc/book/glossary/>

Groovy syntax

<https://groovy-lang.org/semantics.html>

► Quiz

Describing Jenkins and Pipelines

Choose the correct answers to the following questions:

- 1. **Which option best describes a pipeline?**
 - a. A pipeline is a series of steps executed by an automation tool.
 - b. A pipeline is a series of stages executed by a CI/CD tool.
 - c. A pipeline is a series of connected steps executed to accomplish a task.
 - d. A pipeline is series of stages connected to steps, and executed by an automation tool.
- 2. **Which three of the following are the benefits of using pipelines in your development flow? (Choose three.)**
 - a. Reduces the time-to-market
 - b. Improves the reputation of the company
 - c. Improves the code quality by adding testing stages
 - d. Reduces the number of repetitive tasks
 - e. Reduces the complexity of the project
- 3. **Which option best describes the Jenkins project?**
 - a. Jenkins is a CI tool written in Java and licensed as open source.
 - b. Jenkins is an automation engine that only supports pipelines.
 - c. Jenkins is a CI/CD tool that only supports pipelines.
 - d. Jenkins is an automation engine that supports multiple automation patterns.
- 4. **Which two of the following are advantages of using Jenkins to automate the execution of your pipelines? (Choose two.)**
 - a. Jenkins is an open source project.
 - b. Jenkins is one of the most popular automation tools.
 - c. Jenkins is easy to extend by using plug-ins.
 - d. Jenkins is easy to maintain because it is a Java application.
- 5. **Which option best describes the logical components of a Jenkins architecture?**
 - a. The nodes process work, scheduled by the agents.
 - b. The agents manage the nodes, and the work that nodes process.
 - c. The controller manages the agents, and the work they process.
 - d. The controller schedules the work processed by the agents in stand-alone mode.

► **6. Which element, or elements allows you to build flexible pipelines in Jenkins?**

- a. The Jenkins core functionalities
- b. The Jenkins Pipeline plug-in
- c. The Jenkins controller
- d. The Jenkins controller, and agents
- e. The open source license

► Solution

Describing Jenkins and Pipelines

Choose the correct answers to the following questions:

- 1. **Which option best describes a pipeline?**
 - a. A pipeline is a series of steps executed by an automation tool.
 - b. A pipeline is a series of stages executed by a CI/CD tool.
 - c. A pipeline is a series of connected steps executed to accomplish a task.
 - d. A pipeline is series of stages connected to steps, and executed by an automation tool.
- 2. **Which three of the following are the benefits of using pipelines in your development flow? (Choose three.)**
 - a. Reduces the time-to-market
 - b. Improves the reputation of the company
 - c. Improves the code quality by adding testing stages
 - d. Reduces the number of repetitive tasks
 - e. Reduces the complexity of the project
- 3. **Which option best describes the Jenkins project?**
 - a. Jenkins is a CI tool written in Java and licensed as open source.
 - b. Jenkins is an automation engine that only supports pipelines.
 - c. Jenkins is a CI/CD tool that only supports pipelines.
 - d. Jenkins is an automation engine that supports multiple automation patterns.
- 4. **Which two of the following are advantages of using Jenkins to automate the execution of your pipelines? (Choose two.)**
 - a. Jenkins is an open source project.
 - b. Jenkins is one of the most popular automation tools.
 - c. Jenkins is easy to extend by using plug-ins.
 - d. Jenkins is easy to maintain because it is a Java application.
- 5. **Which option best describes the logical components of a Jenkins architecture?**
 - a. The nodes process work, scheduled by the agents.
 - b. The agents manage the nodes, and the work that nodes process.
 - c. The controller manages the agents, and the work they process.
 - d. The controller schedules the work processed by the agents in stand-alone mode.

► **6. Which element, or elements allows you to build flexible pipelines in Jenkins?**

- a. The Jenkins core functionalities
- b. The Jenkins Pipeline plug-in
- c. The Jenkins controller
- d. The Jenkins controller, and agents
- e. The open source license

► Guided Exercise

Configuring a Developer Environment

In this exercise you will install all the required software for the course.



Note

The versions installed in your device might differ from the examples.

Outcomes

You should be able to:

- Install Python
- Install Node.js
- Install the Java Development Kit (JDK)
- Install Git
- Fork and clone the sample code
- Create a GitHub personal access token
- Install Jenkins in a shared cluster

Before You Begin

To perform this exercise, ensure you have an account on your device with administrator privileges.

Instructions

► 1. Install Python.

- 1.1. Install Python in Red Hat Enterprise Linux, CentOS or Fedora.

Python 3 is installed by default in the latest versions of Red Hat Enterprise Linux, CentOS and Fedora.

Run the `python3 --version` command to verify that the version available in your device is 3.6.8 or later.

```
[user@host ~]$ python3 --version
Python 3.6.8
```

Use the `alternatives` command to configure the `/usr/bin/python` binary to point to the correct python version. The `alternatives` command displays all the python versions available in your device, and allow you to choose which one to use by default. The command might prompt for your password to configure the binding.

```
[user@host ~]$ sudo alternatives --config python
```

...output omitted...

There are 2 programs which provide 'python'.

Selection	Command
.....	
*+ 1	/usr/libexec/no-python
2	/usr/bin/python3

Enter to keep the current selection[+], or type selection number: **2**

Run the python binary with the `--version` option to verify the correct bind of the `/usr/bin/python` binary.

```
[user@host ~]$ python --version
Python 3.6.8
```

1.2. Install Python in Ubuntu.

Python 3 is installed by default in Ubuntu 20.04 LTS.

Run the `python3 --version` command to verify that the version available in your device is 3.6.8 or later.

```
[user@host ~]$ python3 --version
Python 3.8.2
```

1.3. Install Python in macOS.

Python 2 is installed by default in the latest versions of macOS, the required version for this course is 3.6.8 or later. A good approach to manage different Python versions is by using `pyenv`.

Use Homebrew to install `pyenv`.

```
[user@host ~]$ brew install pyenv
Updating Homebrew...
...output omitted...
==> Pouring pyenv-1.2.20.catalina.bottle.tar.gz
/usr/local/Cellar/pyenv/1.2.20: 708 files, 2.5MB
```



Note

For more information about the use and installation of Homebrew, refer to the project documentation at <https://docs.brew.sh>.

Use the `pyenv` command to install Python 3.6.8.


```
[user@host ~]$ pyenv install 3.6.8
...output omitted...
Downloading Python-3.6.8.tar.xz...
...output omitted...
Installed Python-3.6.8 to /Users/your-user/.pyenv/versions/3.6.8
```

Use the `pyenv` command to configure the default Python version to be used by your device.

```
[user@host ~]$ pyenv global 3.6.8
```

Add the following code to the end of your `~/.zshrc` file to enable auto-completion. Create the file if it does not exist.

```
if command -v pyenv 1>/dev/null 2>&1; then
    eval "$(pyenv init -)"
fi
```



Note

You might need to restart the shell to use the newly installed Python version.

Run the `python` binary with the `--version` option to verify that the version available in your device is `3.6.8`.

```
[user@host ~]$ python --version
Python 3.6.8
```

1.4. Install Python in Windows.

Open a web browser, navigate to <https://www.python.org/downloads> and download the Windows installer.

This course requires the installation of Python 3.6 or higher. At the moment of creation of this course, the latest version of Python available for Windows is 3.9.

Run the installer, select the **Add Python 3.9 to PATH** check box, click **Install Now**, and follow the installation setup prompts.

Open up Windows PowerShell and verify Python version 3.9 or higher is installed.

```
PS C:\Users\user> python --version
Python 3.9.1
```

2. Install Node.js.

2.1. Install Node.js in Red Hat Enterprise Linux, CentOS or Fedora.

Use the `dnf` command to install Node.js.

```
[user@host ~]$ sudo dnf module install nodejs:14 -y
...output omitted...
Installed:
  nodejs-1:14.11.0-1.module_el8.3.0+516+516d0fc0.x86_64  ...output omitted...

Complete!
```

**Note**

If you need alternate installation instructions, see the options provided here:
<https://nodejs.org/en/download/package-manager/#centos-fedora-and-red-hat-enterprise-linux>.

Verify Node.js version 14 or higher is installed. You might need to restart your shell session beforehand.

```
[user@host ~]$ node -v
v14.11.0
```

2.2. Install Node.js in Ubuntu.

Install the official binary Node.js repositories.

```
[user@host ~]$ sudo su -c \
"bash <(wget -qO- https://deb.nodesource.com/setup_14.x)"

## Installing the NodeSource Node.js 14.x repo...

...output omitted...
```

Use the `apt-get` command to install Node.js.

```
[user@host ~]$ sudo apt-get install -y nodejs
...output omitted...
Setting up nodejs (14.15.3-deb-1nodesource1) ...
Processing triggers for man-db (2.9.1-1) ...
```

Verify Node.js version 14 or higher is installed.

```
[user@host ~]$ node -v
v14.15.3
```

2.3. Install Node.js in macOS.

Install via Homebrew.

```
[user@host ~]$ brew install nodejs
==> Downloading https://homebrew.bintray.com/bottles/
node-15.4.0.catalina.bottle.tar.gz
...output omitted...
==> Pouring node-15.4.0.catalina.bottle.tar.gz
# /usr/local/Cellar/node/15.4.0: 3,273 files, 55.4MB
```

The exact version downloaded will vary based on your machine. You need Node.js version 14 or higher for this course.



Note

Homebrew is an open source package manager for macOS. Although unofficial, it is a common tool used by developers.

Installation instructions for Homebrew can be found on their main page: <https://brew.sh/>

Alternate Node.js downloads for macOS can be found here: <https://nodejs.org/en/download/>

Verify Node.js version 14 or higher is installed. You might need to restart your shell session beforehand.

```
[user@host ~]$ node -v
v15.4.0
```

2.4. Install Node.js on Windows 10.

In a web browser, visit <https://nodejs.org/en/download/> and download the Windows installer.

Run the installer and follow the installation setup prompts.

Close your currently opened Windows PowerShell terminal, open up a new one, and verify that Node.js version 14 or higher is installed.

```
PS C:\Users\user> node -v
v14.17.1
```

► 3. Install the Java Development Kit (JDK).

3.1. Install the JDK in Red Hat Enterprise Linux, CentOS, or Fedora.

Run the `java -version` command to verify that Java is not installed in your system.

```
[user@host ~]$ java -version
bash: java: command not found...
```

If Java 11 or later is not found in your system, then install OpenJDK 11 via `dnf`.

```
[user@host ~]$ sudo dnf install java-11-openjdk-devel -y
...output omitted...
Installed:
  java-11-openjdk-devel-1:11.0.5.10-0.el8_0.x86_64    ...output omitted...

Complete!
```

Run the `java -version` command to verify that the version available in your device is 11 or later.

```
[user@host ~]$ java -version
openjdk 11.0.9-ea 2020-10-20
OpenJDK Runtime Environment 18.9 (build 11.0.9-ea+6)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.9-ea+6, mixed mode, sharing)
```



Warning

If you have an old JDK version installed in your system, e.g. OpenJDK 1.8, then use the `sudo alternatives --config java` command to switch to the newly installed version.

3.2. Install the JDK in Ubuntu.

Run the `java -version` command to verify that Java is not installed in your system.

```
[user@host ~]$ java -version
bash: java: command not found...
```

If Java 11 or later is not found in your system, then install OpenJDK via `apt`.

```
[user@host ~]$ sudo apt install openjdk-11-jdk -y
...output omitted...
Setting up openjdk-11-jre-headless:amd64 (11.0.9.1+1-0ubuntu1~20.04) ...
...output omitted...
```

Run the `java -version` command again to verify that the version installed in your device is 11 or later.

```
[user@host ~]$ java -version
openjdk 11.0.9.1 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.04)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.04, mixed mode, sharing)
```



Warning

If you have an old JDK version installed in your system, e.g. OpenJDK 1.8, then use the `sudo alternatives --config java` command to switch to the newly installed version.

3.3. Install the JDK in macOS.

Install via Homebrew.

```
[user@host ~]$ brew install java11
...output omitted...
==> Summary
# /usr/local/Cellar/openjdk@11/11.0.10: 654 files, 297.3MB
```

Create a symlink in `/Library/Java/JavaVirtualMachines` to help your system discover the installed version.

```
[user@host ~]$ sudo ln -sfn \
/usr/local/opt/openjdk@11/libexec/openjdk.jdk \
/L library/Java/JavaVirtualMachines/openjdk-11.jdk
```

Use the `java_home` command to check that your system detects the newly installed version.

```
[user@host ~]$ /usr/libexec/java_home -V
Matching Java Virtual Machines (3):
  1.8.0, x86_64: "OpenJDK 1.8.0" /Library/Java/JavaVirtualMachines/
openjdk-1.8.0_181/Contents/Home
  11.0.9, x86_64: "OpenJDK 11.0.9" /Library/Java/JavaVirtualMachines/
openjdk-11.jdk/Contents/Home
  11.0.7, x86_64: "GraalVM CE 20.1.0" /Library/Java/JavaVirtualMachines/
graalvm-ce-java11-20.1.0/Contents/Home
...output omitted...
```

If you had a previously installed version, then you must switch to the new version.

Prepend the Java path to your `PATH` variable.

```
[user@host ~]$ echo 'export PATH="/usr/local/opt/openjdk@11/bin:$PATH"' >>
~/.bashrc
```

Reload your shell

```
[user@host ~]$ exec $SHELL
```

Run the `java -version` command to verify that the version available in your device is 11 or later.

```
[user@host ~]$ java -version
openjdk 11.0.9 2020-10-20
OpenJDK Runtime Environment (build 11.0.9+11)
OpenJDK 64-Bit Server VM (build 11.0.9+11, mixed mode)
```



Note

If you need alternate installation instructions, see the options provided here: <https://adoptopenjdk.net/installation.html>.

3.4. Install the JDK in Windows.

Open a browser and navigate to <https://adoptopenjdk.net>.

In the **Choose a Version** area, click **OpenJDK 11 (LTS)**.

In the **Choose a JVM** area, click **HotSpot**.

Click **Latest release** to download the Windows installer.

Run the installer and follow the installation setup prompts.



Warning

Make sure to select the option **Set JAVA_HOME variable** during the installation.

Close your currently opened Windows PowerShell terminal, open up a new one, and verify the installation.

```
PS C:\Users\user> java -version
openjdk 11.0.9.1 2020-11-04
...output omitted...
```

► 4. Install Git.

4.1. Install Git in Red Hat Enterprise Linux or Fedora.

Use the `dnf` command to install the `git` package. The command might prompt for your password.

```
[user@host ~]$ sudo dnf install git -y

...output omitted...

Installing:
git                x86_64 2.18.4-2.el8_2  rhel-8-for-x86_64-appstream-rpms 187 k

...output omitted...

Installed:
git-2.18.4-2.el8_2.x86_64          git-core-2.18.4-2.el8_2.x86_64
git-core-doc-2.18.4-2.el8_2.noarch perl-Error-1:0.17025-2.el8.noarch
perl-Git-2.18.4-2.el8_2.noarch    perl-TermReadKey-2.37-7.el8.x86_64

Complete!
```

Run the `git --version` command to verify the correct installation of the `git` package.

```
[user@host ~]$ git --version
git version 2.18.4
```

4.2. Install Git in Ubuntu.

Use the `apt-get` command to install the `git` package. The command might prompt for your password.

```
[user@host ~]$ sudo apt-get install -y git
Reading package lists... Done
...output omitted...
Setting up git (1:2.25.1-1ubuntu3) ...
Processing triggers for man-db (2.9.1-1) ...
```

Run the `git --version` command to verify git version 2.18 or higher is installed.

```
[user@host ~]$ git --version
git version 2.25.1
```

4.3. Install Git in macOS.

Git is installed by default in the latest versions of macOS.

Run the `git --version` command in a command line terminal to verify that the version available in your device is 2.18 or later.

```
[user@host ~]$ git --version
git version 2.24.1 (Apple Git-126)
```

To install a newer version of git, use Homebrew:

```
[user@host ~]$ brew install git
==> Downloading https://homebrew.bintray.com/bottles/
git-2.28.0.catalina.bottle.tar.gz
...output omitted...
==> Summary
# /usr/local/Cellar/git/2.28.0: 1,482 files, 48.9MB
```

4.4. Install Git in Windows 10.

Open a web browser, navigate to <https://git-scm.com/download/win> and download the Windows installer.

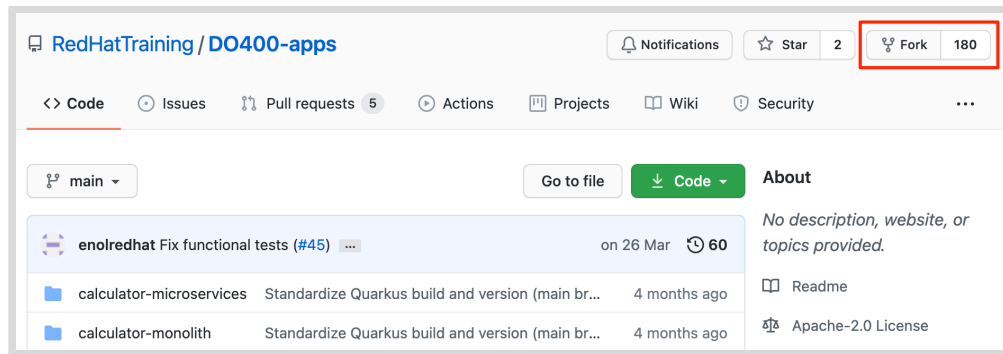
Run the installer and follow the installation setup prompts.

Close your currently opened Windows PowerShell terminal, open up a new one, and verify that Git version 2.18 or higher is installed.

```
PS C:\Users\user> git --version
git version 2.30.0.windows.1
```

- ▶ 5. For some of the exercises, you need the sample applications of this course. Fork the `D0400-apps` repository to your personal GitHub account.

- 5.1. Open a web browser and navigate to <https://github.com/RedHatTraining/DO400-apps>. If you are not logged in to GitHub, then click **Sign in** in the upper-right corner.
- 5.2. Log in to GitHub by using your personal user name and password.
- 5.3. Return to the `RedHatTraining/DO400-apps` repository and click **Fork** in the upper-right corner.



- 5.4. In the Fork **DO400-apps** window, click **YOUR_GITHUB_USER** to select your personal GitHub account.
- 5.5. After a few seconds, the GitHub web interface displays your new repository **YOUR_GITHUB_USER/DO400-apps**.
- ▶ 6. Before starting exercises that require sample applications, you also must clone the code of the **YOUR_GITHUB_USER/DO400-apps** repository to your workstation.
 - 6.1. Create a workspace folder and change to the newly created folder. The exercises in this course use the `~/D0400` workspace folder but you can choose a different one.

```
[user@host ~]$ mkdir D0400
[user@host ~]$ cd D0400
[user@host D0400]$
```

- 6.2. Run the following command to clone this course's sample applications repository to your workstation. Replace **YOUR_GITHUB_USER** with the name of your personal GitHub account:

```
[user@host D0400]$ git clone https://github.com/YOUR_GITHUB_USER/DO400-apps
Cloning into 'D0400-apps'...
...output omitted...
```

- 6.3. Verify that the cloned repository contains the sample application and return to the workspace directory.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ head README.md
# D0400 Application Repository
...output omitted...
[user@host D0400-apps]$ cd ..
[user@host D0400]$
```



Important

Windows users must replace the preceding `head` command in PowerShell as follows:

```
PS C:\Users\user\D0400\D0400-apps> type README.md
```


- 7. Create a GitHub personal access token, if you do not already have one.

GitHub has deprecated password-based authentication. This means that you must use a personal access token to run Git commands that require authentication, such as cloning private repositories or pushing changes to GitHub.

If you do not already have a personal access token, then use the [Creating a personal access token](https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token) [https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token] guide to create a new one.

Some exercises in this course will require you to authenticate to GitHub. When asked for your password, enter your personal access token.



Note

SSH-based authentication does not require a personal access token.

However, the exercises in this course cover HTTPS-based authentication, which means that you need a personal access token. If you are comfortable with and prefer SSH-based authentication with GitHub, be aware that the URLs used in this course will not work properly for you.

- 8. Install the OpenShift command line interface (CLI).

- 8.1. Download the CLI application through the OpenShift web console. Find the **Lab Environment** tab on your **ROLE** course page. This table should be visible after you provision your online lab environment.

Table of Contents Course Lab Environment ★ ?

► Lab Controls

Click **CREATE** to build all of the virtual machines needed for the classroom lab environment. This may take several minutes to complete. Once created the environment can then be stopped and restarted to pause your experience.

If you **DELETE** your lab, you will remove all of the virtual machines in your classroom and lose all of your progress.

DELETE **STOP** ⓘ

OpenShift Details

Username	RHT_OCP4_DEV_USER	youruser
Password	RHT_OCP4_DEV_PASSWORD	yourpassword
API Endpoint	RHT_OCP4_MASTER_API	https://api.cluster.domain.example.com:6443
Console Web Application	https://console-openshift-console.apps.cluster.domain.example.com	
Cluster Id	your-cluster-id	

workstation	active	ACTION ~	OPEN CONSOLE
classroom	active	ACTION ~	OPEN CONSOLE

The **Console Web Application** link in the **OpenShift Details** table opens the web console for your dedicated Red Hat OpenShift Container Platform instance. Take note of your **Username** and **Password**, which you will be using in the next step.

- 8.2. Log in to the OpenShift web console using your **Username** and **Password**.
- 8.3. After logging in, click ? in the upper-right corner, and then click **Command Line Tools**.

- 8.4. On the **Command Line Tools** page, download the OpenShift CLI compressed binary file for your platform from the list **oc - OpenShift Command Line Interface (CLI)**
- 8.5. Unpack the compressed archive file, and then copy the **oc** binary to a directory of your choice. Ensure that this directory is in the **PATH** variable for your system. On macOS and Linux, copy the **oc** binary to `/usr/local/bin`:

```
[user@host ~]$ sudo cp oc /usr/local/bin/
[user@host ~]$ sudo chmod +x /usr/local/bin/oc
```

On Windows 10 systems, decompress the downloaded archive, and then follow the instructions at <https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/>. Edit the **PATH** environment variable and add the full path to the directory where the **oc** binary is located.

- 8.6. Verify that the **oc** binary works for your platform. Open a new command line terminal and run the following:

```
[user@host ~]$ oc version --client
Client Version: 4.6.0
```

**Note**

Your output might be slightly different based on the version of the OpenShift client that you downloaded.

**Note**

If you are using macOS, then you might need to manually allow the **oc** command to run. If so, open **System Preferences**, click **Security & Privacy**, and allow the unverified developer. This will only show up after attempting to run the command.

- ▶ 9. Deploy a Jenkins instance on OpenShift. To log in to the cluster, use the values given to you in the **Lab Environment** tab, after you provision your online lab environment:

Table of Contents Course Lab Environment ★ ⓘ

► Lab Controls

Click **CREATE** to build all of the virtual machines needed for the classroom lab environment. This may take several minutes to complete. Once created the environment can then be stopped and restarted to pause your experience.

If you **DELETE** your lab, you will remove all of the virtual machines in your classroom and lose all of your progress.

DELETE **STOP** ⓘ

OpenShift Details

Username	RHT_OCP4_DEV_USER	youruser
Password	RHT_OCP4_DEV_PASSWORD	yourpassword
API Endpoint	RHT_OCP4_MASTER_API	https://api.cluster.domain.example.com:6443
Console Web Application		https://console-openshift-console.apps.cluster.domain.example.com
Cluster Id		your-cluster-id

workstation	active	ACTION ▾	OPEN CONSOLE
classroom	active	ACTION ▾	OPEN CONSOLE

9.1. Log in to OpenShift by using your developer user account.

```
[user@host ~]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
```



Important

Windows users must adapt line-breaking characters in multi-line commands.

In PowerShell, use the backtick character (`) to introduce a line break in a command. For example, to run the preceding command in Windows PowerShell, you must run it as follows:

```
PS C:\Users\user> oc login -u RHT_OCP4_DEV_USER `
>> -p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
```

Note that PowerShell prints the >> prompt as a line-continuation marker.

This course does not provide support for alternative Windows shells, such as the Windows cmd command interpreter.

9.2. Create a new project to host the Jenkins instance. Prefix the project name with your developer user name.

```
[user@host ~]$ oc new-project RHT_OCP4_DEV_USER-jenkins
Now using project "youruser-jenkins" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

9.3. Browse the Jenkins templates available in OpenShift.

```
[user@host ~]$ oc get templates -n openshift | grep jenkins
jenkins-ephemeral      Jenkins service, without persistent storage...
jenkins-ephemeral-monitored  Jenkins service, without persistent storage. ...
jenkins-persistent     Jenkins service, with persistent storage...
jenkins-persistent-monitored  Jenkins service, with persistent storage. ...
```



Important

Windows users must also run the preceding command but in PowerShell as follows:

```
PS C:\Users\user> oc get templates -n openshift | findstr /R /C:"jenkins"
```

For this course, you will use the `jenkins-ephemeral` template.

9.4. Deploy Jenkins on OpenShift.

```
[user@host ~]$ oc new-app \
openshift/jenkins-ephemeral -p MEMORY_LIMIT=2048Mi
--> Deploying template "openshift/jenkins-ephemeral" to project youruser-ci-cd
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```



Important

Windows PowerShell users must use the backtick character (```) instead of the backslash character (`\`) to introduce line breaks in the preceding command.

The Jenkins instance will take some time to deploy.



Note

The `MEMORY_LIMIT` parameter increases the amount of memory allocated to the Jenkins container to improve performance. The JVM maximum heap size is set to 50% of the value of `MEMORY_LIMIT` by default. The default value of `MEMORY_LIMIT` is 1GB.

9.5. Get the URL of the Jenkins instance by inspecting the route.

```
[user@host ~]$ oc get route jenkins
NAME          HOST/PORT          ...output omitted...
jenkins       youruser-jenkins.apps.cluster.domain.example.com ...output omitted...
```

- 9.6. Open the URL in a web browser. To log in, use the OpenShift credentials provided to you when you created the lab environment. You should see the Jenkins main page.

This concludes the guided exercise.

Summary

In this chapter, you learned:

- Continuous Integration incorporates changes in the main branch as often as possible. Automated integration tests validate the integrated changes.
- Continuous Delivery reduces the manual process involved in the creation of application releases. In this practice, the release deployment requires human intervention.
- Continuous Deployment performs the release deployment automatically, without human intervention.
- A pipeline is a series of connected steps executed in sequence to accomplish a task.
- Jenkins is an automation engine that extends its capabilities with plug-ins.
- The Jenkins environment consists of a controller, and one or more agents.

Chapter 2

Integrating Source Code with Version Control

Goal

Manage source code changes with Git version control.

Objectives

- Define the role of version control in Continuous Integration.
- Contribute to application source code with Git.
- Create feature branches with Git.
- Manage and collaborate with remote repositories.
- Version and publish code with Git.

Sections

- Introducing Version Control (and Quiz)
- Building Applications with Git (and Guided Exercise)
- Creating Git Branches (and Guided Exercise)
- Managing Remote Repositories (and Guided Exercise)
- Releasing Code (and Guided Exercise)

Lab

Integrating Source Code with Version Control

Introducing Version Control

Objectives

After completing this section, you should be able to define the role of version control in Continuous Integration.

Describing Version Control

Software engineering is inherently collaborative, and organizations often distribute developers amongst teams working on multiple projects. With so many developers working on shared code bases, they need effective collaboration tools and techniques. Both of these reduce accidental overwrites and bugs, while promoting good integration habits.

Version Control is the practice of managing the changes that happen in a set of files or documents over time. In software engineering, *Version Control Systems* (VCS) are applications that store, control, and organize the changes that happen in the source code of an application.

With version control systems, teams can **organize and coordinate** their work more reliably. When several developers are working on the same project simultaneously, they must manage changes in a way that avoids overriding others' work, and solve potential conflicts that arise.

Version control systems serve as a centralized, shared, or distributed storage for teams to save their progress.

To coordinate work, teams store their code base in *repositories*. A repository is a common location within a VCS where teams store their work. Source code repositories store the project files, as well as the history of all changes made to those files and their related metadata.

Normally, developers work with multiple repositories in their VCS. A common approach is to use a different repository for each project.

Teams are free to choose which source files they want to include in the version control system. For example, teams can decide to use version control to store and manage the configuration of their platform infrastructure. This practice is known as *Infrastructure as Code* (IaC) and is a critical part of building a DevOps culture.

There are multiple implementations of version control systems available. The most popular version control implementation is Git, which is the VCS that is used throughout this course. Other popular version control systems are Mercurial, SVN, and CVS.

Defining the Role of Version Control in Continuous Integration

Version Control is a requirement for adopting other DevOps techniques. In particular, Version Control is the foundation for Continuous Integration (CI). Because Continuous Integration is about integrating changes to files across a team frequently, you need a place to store the changes. By using a VCS, you use repositories as the common location where you store your work and integrate changes.

Developers use *branches* as a way to develop their work without interfering with other team members' work. In the context of Version Control, a branch is a specific stream of work in a repository.

Repositories usually have a principal or *main branch*, as well as a series of secondary *feature branches* where developers carry out their work. After developers finish their work in a feature branch and the code is reviewed by another team member, they integrate the changes of the feature branch into the main branch. This basic workflow is the foundation for continuous integration.

It is critical to ensure that all developers are using Continuous Integration because the longer it takes for a branch to be integrated into the code base, the higher the likelihood there will be conflicts with other developers' work. In addition to integrating into the repository frequently, a critical tenet of Continuous Integration is that those changes should be validated before they are integrated. To properly validate changes, run the following integration tests:

- **Preintegration local checks:** before pushing changes to the repository, the developer runs a set of tests to validate that the changes to be pushed are correct. Developers run these tests in their local development environments.
- **Preintegration branch checks:** when a developer pushes changes to a feature branch, an automation server runs integration tests in the context of this branch. The server normally pulls the code from the feature branch and executes a pipeline to test that the branch passes the tests. If tests pass and the team agrees on integrating the changes, then the feature branch is integrated into the main branch.
- **Postintegration checks:** after the developer integrates changes into the main branch, the integration pipeline runs. The automation server pulls the code from the main branch and executes the integration pipeline to validate that the integration has been successful. This step ensures that the main branch remains correct after integrating changes.

To follow this process you need an automation server, such as Jenkins, with access to the VCS to pull the code. Additionally, there are different options to configure how the server triggers the execution of pipelines.

- **Manual:** After a developer pushes a change to the repository, either the same developer or another team member triggers the execution of the CI pipeline in the automation server. This approach requires human intervention.
- **Polling:** The automation server repeatedly checks the VCS for changes in the repository. For example, every 5 minutes. When a change is detected, the server runs the integration pipeline.
- **Hooks:** The team registers a hook in the VCS to notify the automation server about changes. When the repository changes, the VCS posts a notification to the automation server. The automation server reacts to this notification by running the CI pipeline. This approach aligns with DevOps principles, automatically triggering a CI pipeline right after a repository changes.

Version Control Forges

The easiest option to adopt version control with DevOps features in mind is to use a version control forge. Version control forges are applications that wrap the core functionality of a VCS system, such as Git, with additional features around the version control core.

These applications normally try to support a DevOps stack, offering friendly web interfaces, which expose features such as code reviews, bug tracking, agile project management, and integration with third-party systems.

Many of the forges also include advanced features, such as full-fledged CI/CD automation modules, and integration with monitoring systems. This turns these forges into all-in-one solutions for DevOps tooling.

GitHub is arguably the most popular forge, although other applications such as GitLab, Bitbucket, and Pagure are also popular. You will use GitHub in this course.



References

Version control

https://en.wikipedia.org/wiki/Version_control

Comparison of source-code-hosting facilities

https://en.wikipedia.org/wiki/Comparison_of_source-code-hosting_facilities

What is Git?

<https://opensource.com/resources/what-is-git>

► Quiz

Introducing Version Control

Choose the correct answers to the following questions:

- 1. **Which two of the following are Version Control Systems (VCS)? (Choose two.)**
 - a. Mercurial
 - b. GitLab
 - c. GitHub
 - d. Git

- 2. **How does version control primarily help the development process?**
 - a. By monitoring the application performance.
 - b. By running an automation server for Continuous Integration.
 - c. By storing and tracking the changes in a project over time.
 - d. By executing Continuous Integration pipelines.

- 3. **You have configured a CI pipeline to run integration tests and now you want your automation server to react to changes in your repository. What is the best option to make the automation server run the CI pipeline after a repository change?**
 - a. Register a hook in the version control system to send change notifications to the automation server.
 - b. Let the team members decide when to run the pipeline after changes have been integrated into the main branch.
 - c. Configure the automation server to continuously monitor the repository.
 - d. Configure the automation server to run the CI pipelines every minute.

- 4. **Which of the following are recommended ways to check that changes integrate correctly into the main branch?**
 - a. Before committing or pushing changes, make sure that tests pass in your local environment.
 - b. Before integrating changes from the feature branch into the main branch, check that the automation server has successfully tested the feature branch.
 - c. After integrating the changes into the main branch, check that the automation server has successfully tested the main branch.
 - d. All of the above.

► Solution

Introducing Version Control

Choose the correct answers to the following questions:

- 1. **Which two of the following are Version Control Systems (VCS)? (Choose two.)**
 - a. Mercurial
 - b. GitLab
 - c. GitHub
 - d. Git

- 2. **How does version control primarily help the development process?**
 - a. By monitoring the application performance.
 - b. By running an automation server for Continuous Integration.
 - c. By storing and tracking the changes in a project over time.
 - d. By executing Continuous Integration pipelines.

- 3. **You have configured a CI pipeline to run integration tests and now you want your automation server to react to changes in your repository. What is the best option to make the automation server run the CI pipeline after a repository change?**
 - a. Register a hook in the version control system to send change notifications to the automation server.
 - b. Let the team members decide when to run the pipeline after changes have been integrated into the main branch.
 - c. Configure the automation server to continuously monitor the repository.
 - d. Configure the automation server to run the CI pipelines every minute.

- 4. **Which of the following are recommended ways to check that changes integrate correctly into the main branch?**
 - a. Before committing or pushing changes, make sure that tests pass in your local environment.
 - b. Before integrating changes from the feature branch into the main branch, check that the automation server has successfully tested the feature branch.
 - c. After integrating the changes into the main branch, check that the automation server has successfully tested the main branch.
 - d. All of the above.

Building Applications with Git

Objectives

After completing this section, you should be able to contribute to application source code with Git.

Introducing Git

Git is a *version control system* (VCS) originally created by Linus Torvalds in 2005. His goal was to develop a version control system (VCS) for maintaining the Linux kernel. He cites the main design philosophy behind Git is a focus on being distributed. Projects are tracked in Git repositories. Within that repository, you run Git commands to create a snapshot of your project at a specific point in time. In turn, these snapshots are organized into branches.

The snapshots and branches are typically uploaded ("pushed") to a separate server ("remote"). From there, any number of workflows can be used to organize code. For this course, we will only focus on a widely used workflow based on "pull requests".

Distributed vs Centralized

In a *centralized* VCS, such as Subversion, you must upload file changes to a central server. The server's copy of the repository is considered the "single source of truth". Although a centralized model is easier to understand, it carries limitations in workflow flexibility.

In a *decentralized* VCS, every copy of the repository is a complete or "deep" copy. In a decentralized system, the designation of "primary" is arbitrary. Consequently, any copy or even multiple copies could be designated as the primary repository. This includes cloned copies of the repository on your development machine.



Note

Although Git repositories do not *require* a central server or copy, it is common for software projects to designate a primary copy. This is often referred to as the "upstream" repository.

Benefits of a decentralized VCS

- Works offline
- Create local backups
- Flexible workflows
- Ad hoc code sharing
- More granular permissions

Drawbacks of a decentralized VCS

- Steeper learning curve
- Longer onboarding times

- Increased workflow complexity

What is a commit?

As you make changes to files within your repository, you can persist those changes in a commit. A *commit* stores file changes and the values displayed in the following table.

Git also tracks commit order by storing a reference to a parent commit within each commit. This way, as you create commits, they form a chain.



Figure 2.1: Relation between commits

You can imagine each commit as a new layer of changes on top of the last. This approach allows you to more easily rollback and track changes or catch up to new changes in files.

To recreate the files as they were in a *previous state*, Git applies the commits up to the desired point. For example, you can roll back a file to a previous working state in the case that a new commit introduced a bug in your application.

Contents of a Commit

Name	Purpose
Author	Name and email address of the person who committed the changes.
Time stamp	The date and time of commit creation.
Message	A brief comment describing changes made.
Hash	A generated unique identifier (UID) for the commit.

Most of these values are calculated by Git or they are preconfigured. For example, you typically configure your name and email by using the `git config` subcommand, which Git will then store in any commits you create.

When creating a commit, you *must* provide a message. Although the message only needs to be a few words, it should be descriptive. Often, commit messages are crucial to understanding the history of the repository. A good commit message describes the changes made in the commit, and the context of those changes. That way, other developers (or your future self) can easily know what was done without the need to inspect the code changes.

The commit hash is calculated from other commit metadata, such as the author, commit message, and time stamp. Because it uses these values and SHA-1 to calculate the hash value, collisions are rare.

Commit history is not always linear as you will see later, making helpful commit messages even more valuable.

Managing Git Repositories

The primary interface for Git is the provided command-line interface (CLI). This CLI includes various "subcommands", such as `add`, `init`, and `commit`. Git provides a man page for each of these subcommands. For example, you can view the man page for the `add` subcommand by running:

```
[user@host ~]$ git help add
```

Obtaining a Git Repository

To run most Git operations, you must run the command within a Git repository.

One option is to initialize a repository in your local system by using `git init`. This subcommand will create a `.git` directory. Remember, this subdirectory contains all of the information about your repository.

If a directory is already a Git repository, then you cannot initialize it as a repository again. This is because a repository is defined as any directory containing a child directory named `.git`. Any other child directories and their contents are considered part of that same repository.



Note

Avoid accidentally nesting Git repositories. This is an advanced technique known as a "submodule". Submodules are not covered in this course.

Another option is to clone an existing repository by using `git clone`. For example, you can clone from GitHub, which is a popular code sharing website.

You will use `git init` to create a new repository in the corresponding guided exercise.

Staging Changes

In order to create a commit, you must first *stage* changes. Any "unstaged" changes will not be included in the commit. Requiring you to first stage changes allows you to control which changes are committed. This level of control is helpful when you want to save part of your changes. For example, when you are developing a big feature, you can do small commits with the parts that you think are complete.

Add a file's changes to the stage with the `git add` subcommand and specify the name of the file. To remove changes from the stage, use `git reset`. Add or remove changed files from the stage individually or recursively by specifying a file or directory, respectively.

Creating a Commit

Once you have staged changes, you can use `git commit` to create a commit. With your changes committed, you are free to continue making changes knowing that you can easily restore to earlier versions. Be sure to "commit early and often" as commits allow you to restore your files to the state captured within.

**Note**

In general, you do not have to worry about a Git repository's storage usage. Commits containing changes to text-based files are very light on storage space.

However, Git is unable to store just the changes to binary files such as pictures. This means Git must store additional copies of the picture every time it is changed.

As you use Git, you adopt the following workflow:

- Make changes to your project files
- Stage changes with the `git add` command
- Commit the changes with the `git commit` command

**References****Git Documentation**

<https://www.git-scm.com/>

***Git Pro* section on staging and committing**

<https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>

Tech Talk: Linus Torvalds on git, 2007

<https://www.youtube.com/watch?v=4XpnKHJAok8>

► Guided Exercise

Building Applications with Git

In this exercise you will configure `git`, create a Python application, and use a code repository to manage the project source.

Outcomes

You should be able to configure `git`, initialize a repository, manage the repository files, commit file changes, and undo changes to the files.

Before You Begin

To perform this exercise, ensure you have Git installed on your computer and access to a command line terminal.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start every guided exercise from this folder.

Instructions

- 1. Configure your git identity at the user level in your machine.



Note

You can skip this step if you already have your git identity setup on your machine.

- 1.1. Open a new terminal window on your workstation, and use the `git config` command to set up your user name.

```
[user@host D0400]$ git config --global user.name "Your User Name"
```

- 1.2. Use the `git config` command to set up your user email.

```
[user@host D0400]$ git config --global user.email your@user.email
```

- 1.3. Use the `git config` command to review your identity settings.

```
[user@host D0400]$ git config --list
user.name=Your User Name
user.email=your@user.email
```

- 2. Create a folder for the exercise files and then initialize a Git repository.

- 2.1. Create a folder named `git-basics` in your workspace directory.

```
[user@host D0400]$ mkdir git-basics
```

2.2. Navigate to the `git-basics` folder on the command terminal.

```
[user@host D0400]$ cd git-basics
[user@host git-basics]$
```

2.3. Use the `git init` command to initialize the git repository in the current directory.

```
[user@host git-basics]$ git init .
Initialized empty Git repository in ...output omitted.../D0400/git-basics/.git/
```

- 3. Create a Python application, add it to the repository, and verify the staged changes.
Use a text editor such as VSCodium, which supports syntax highlighting for editing Python source files.

3.1. Create a Python file named `person.py` with the following content.

```
class Person:
    def greet(self) -> None:
        print('Hello Red Hatter!')

pablo = Person()

pablo.greet()
```

3.2. Run the Python script to verify that it greets you as a Red Hatter.

```
[user@host git-basics]$ python person.py
Hello Red Hatter!
```



Warning

In some Python installations, the python binary is `python3` instead of `python`.

3.3. Use the `git add` command to add the new file to the staging area.

```
[user@host git-basics]$ git add person.py
```

3.4. Run the `git status` command to check that the file changes are in the staging area.

```
[user@host git-basics]$ git status
On branch master

No commits yet

Changes to be committed:
```

(use `git rm --cached <file>...` to unstage)

new file: person.py

- 3.5. Use the `git commit` command to commit the staged file, and assign to the commit a meaningful comment with the `-m` option.

```
[user@host git-basics]$ git commit -m "Initial Person implementation"
[master (root-commit) b030555] Initial Person implementation
1 file changed, 7 insertions(+)
create mode 100644 person.py
```

Every time you add a commit to a repository, git generates and associates a hash string to the commit in order to identify it. In this case, the hash string `b030555` was generated and assigned to the initial commit.

Note that the hash strings assigned to your commits might differ from the examples.

- 3.6. Rename the current branch `master` (the default in Git) to `main`

```
[user@host git-basics]$ git branch -M main
```

- 3.7. Run the `git status` command to visualize the current state of the repository.

```
[user@host git-basics]$ git status
On branch main
nothing to commit, working tree clean
```

- ▶ 4. Modify the `person.py` file to improve the greeting feature, review the changes, and commit the new version of the file.

- 4.1. Modify the `person.py` file to allow a personalized greeting and salute to Jaime and Guy. The `person.py` file should look like the following:

```
class Person:
    def greet(self, name: str = 'Red Hatter') -> None:
        print('Hello {}'.format(name))

pablo = Person()

pablo.greet('Jaime')
pablo.greet('Guy')
```

- 4.2. Run the Python script to verify that it salutes Jaime and Guy.

```
[user@host git-basics]$ python person.py
Hello Jaime!
Hello Guy!
```

- 4.3. Run the `git diff` command to view the differences between the original version of the file, and the latest changes.

```
[user@host git-basics]$ git diff
diff --git a/person.py b/person.py
index 0652fa3..7750914 100644
--- a/person.py
+++ b/person.py
@@ -1,7 +1,7 @@
 class Person:
-    def greet(self) -> None:
-        print('Hello Red Hatter!')
+    def greet(self, name: str = 'Red Hatter') -> None:
+        print('Hello {}'.format(name))

pablo = Person()

-pablo.greet()
+pablo.greet('Jaime')
+pablo.greet('Guy')
```

The `git diff` command highlights the changes made to the `person.py` file. The `-` character at the beginning of a line indicates that the line was removed. The `+` character at the beginning of a line indicates that the line was added.



Note

Git uses a pager when the changes to display are too large to fit in the screen. In this pager you can scroll up and down to see the changes. Press `q` to exit the pager.

4.4. Use the `git add` command to add the changes to the staging area.

```
[user@host git-basics]$ git add .
```

4.5. Run the `git status` command to check that the file changes are in the staging area.

```
[user@host git-basics]$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   person.py
```

4.6. Use the `git commit` command to commit the staged changes, and assign to the commit a meaningful comment with the `-m` option.

```
[user@host git-basics]$ git commit -m "Enhanced greeting"
[main b567b44] Enhanced greeting
1 file changed, 4 insertions(+), 3 deletions(-)
```

► 5. Inspect the repository history of commits.

5.1. Run the `git log` command to inspect the commit history of the repository.

```
[user@host git-basics]$ git log
commit b567b44b1fa64c428a199e9776c64fcb99c2b40d (HEAD -> main)
Author: Your User Name <your@user.email>
Date:   Mon Sep 28 11:54:23 2020 +0200

    Enhanced greeting

commit fd86e63c430e5232e028bce4e8998ec3433859bd
Author: Your User Name <your@user.email>
Date:   Mon Sep 28 11:24:07 2020 +0200

    Initial Person implementation
```

The `git log` command shows all the commits of the repository in chronological order. The default output includes the commit hash string, the author, the date of the commit, and the commit message.

- 5.2. Run the `git show` command to view the latest commit, and the changes made in the repository files.

```
[user@host git-basics]$ git show
commit b567b44b1fa64c428a199e9776c64fcb99c2b40d (HEAD -> main)
Author: Your User Name <your@user.email>
Date:   Mon Sep 28 11:54:23 2020 +0200

    Enhanced greeting

diff --git a/person.py b/person.py
index 0652fa3..634d287 100644
--- a/person.py
+++ b/person.py
@@ -1,7 +1,8 @@
 class Person:
-    def greet(self) -> None:
-        print('Hello Red Hatter!')
+    def greet(self, name: str = 'Red Hatter') -> None:
+        print('Hello {}'.format(name))

pablo = Person()

-pablo.greet()
+pablo.greet('Jaime')
+pablo.greet('Guy')
```

- ▶ 6. Modify the `person.py` file to include a question in the greeting feature, stage the changes, and undo all the modifications.
 - 6.1. Modify the `person.py` file to include a question after the greeting. The `person.py` file should look like the following:

```
class Person:
    def greet(self, name: str = 'Red Hatter') -> None:
        print('Hello {}!'.format(name) ' How are you today?')

pablo = Person()

pablo.greet('Sam')
```

Do not run the program yet.

- 6.2. Use the `git add` command to add the file changes to the staging area.

```
[user@host git-basics]$ git add person.py
```

- 6.3. Run the Python script to verify that the question is added after the greeting.

```
[user@host git-basics]$ python person.py
File "person.py", line 3
    print('Hello {}!'.format(name) ' How are you today?')
                                   ^
SyntaxError: invalid syntax
```

The latest changes introduced a bug in our Python script.

- 6.4. The changes that introduced a bug in the Python script are in the staging area. Use the `git reset` command to unstage the changes.

```
[user@host git-basics]$ git reset HEAD person.py
Unstaged changes after reset:
M person.py
```

- 6.5. Run the `git status` command to check that the changes were unstaged.

```
[user@host git-basics]$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   person.py

no changes added to commit (use "git add" and/or "git commit -a")
```

- 6.6. Use the `git checkout` command to undo the local changes of the `person.py` file.

```
[user@host git-basics]$ git checkout -- person.py
```

As covered in *Creating Git Branches*, `git checkout` is normally reserved for traversing the repository, but many of Git's subcommands have alternate uses.

This form of the checkout command restores the `person.py` file to its state at a specified commit or the current commit, by default. The `--` parameter instructs Git to interpret any following arguments as files.

**Note**

Newer versions of Git suggest using `git restore` for this task. However, according to Git documentation, this command is currently experimental.

6.7. Verify that the app continues running fine and return to the workspace directory.

```
[user@host git-basics]$ python person.py
Hello Jaime!
Hello Guy!
[user@host git-basics]$ cd ~/D0400
[user@host D0400]$
```

This concludes the guided exercise.

Creating Git Branches

Objectives

After completing this section, you should be able to create feature branches with Git.

Organizing and Incorporating Changes with Branches

The nature of software development is that the code of applications grows and evolves complexity over time. What was once a small application is now a large project with multiple developers collaborating in the same repository simultaneously.

A project might have different features and fixes in development at the same time. Those streams of work must happen in parallel to optimize the time to market of your project.

With Git, you can organize those streams into collections of code changes named *branches*. Once a stream of work is ready, its branch is merged back into the main branch. Each organization and team should decide how to name their branches. It is important to enforce a strict naming convention to maintain consistency, which makes identifying the contents of a branch much simpler.



Note

Git does not enforce consistent naming, however teams should enforce a convention for various types of branches, such as for bugs, features, and releases.

Establishing a Main Branch

As you work on a repository, it is best to establish a primary or main branch. At any given point, this branch is considered the most recent version of the repository. Any changes to this branch should undergo any necessary quality assurance checks, as outlined in *Introducing Version Control*.

Git allows for high flexibility in workflows, therefore it does not programmatically enforce or ensure any quality metrics of the main branch, or any other branch. This flexibility leaves you with the responsibility of applying quality assurance checks in your repositories.



Note

As of this publication, creating a new Git repository via GitHub creates a default branch named `main`, whereas using the `git init` command produces `master`. Historically, `master` was the default branch name, but many organizations are reconsidering how they can make certain language more inclusive.

You can change the default branch for existing repositories or configure Git to use a custom default branch name.

Defining Branches

Conceptually, a branch houses a collection of commits that implement changes not yet ready for the main branch. Contributors should create commits on a new branch while they work on their changes.

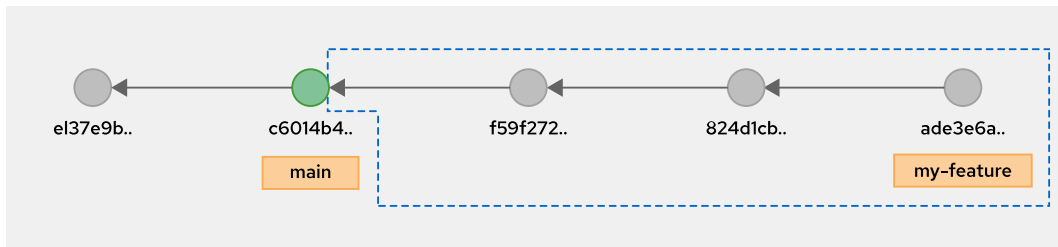


Figure 2.2: Commits in a branch not yet ready for the main branch

In the preceding example, the `my-feature` branch incorporates the commits with hashes `ade3e6a`, `824d1cb`, and `f59f272`. You can quickly compare branches by using `git log`:

- With no arguments, it will show the commit history starting with the current commit.
- If you provide a revision range, then it will show the commits that differ.

For example, in the preceding repository illustration:

```
[user@host repository]$ git log main..my-feature
commit ade3e6a... (HEAD -> my-feature)
Author: User <user@example.com>
Date:   Mon Jan 18 13:12:23 2021 -0500

    most recent commit

commit 824d1cb...
Author: User <user@example.com>
Date:   Fri Jan 15 16:43:39 2021 -0500

    penultimate commit

commit f59f272...
Author: User <user@example.com>
Date:   Thu Jan 14 17:02:21 2021 -0500

    first branch commit
```

Note that it only shows commits that are on the `my-feature` branch, but not ones on `main`.

How Git Stores Branches

Git does not store the contents individually for each branch. Instead, it stores a key-value pair with the name of the branch and a commit hash. For example, in the previous image, Git simply stores the pair: `my-feature: ade3e6a`.

This method of storing branches as references, also known as *pointers*, makes storing and operating on branches computationally trivial. It is one of the fundamental differences between Git and other version control systems, such as Subversion.

Creating a Branch

The main command for managing branches is `git branch`. This command takes a single argument: the name of the new branch. If a branch with the same name already exists, then the command fails.

For example, the following command creates a new branch named `my-branch`:

```
[user@host repository]$ git branch my-branch
```

Note that the `git branch` command only *creates* the branch, it does not switch you to that branch.

Conversely, you can delete a branch by including the `-d` option.

For example, the following would delete the branch named `new-feature`:

```
[user@host repository]$ git branch -d new-feature
Deleted branch new-feature (was ade3e6a).
```

Remember that each branch is stored as a key-value pair and takes up very little storage, so cleaning up old branch names is not especially important.

Navigating Branches

The `git checkout` command is largely responsible for navigating between locations in the repository. It takes a single argument of where you would like to go, usually a branch name.

For example, the following command switches to the `my-bugfix` branch:

```
[user@host repository]$ git checkout my-bugfix
Switched to branch 'my-bugfix'
```

If no branch with the provided name exists, the command fails.

A common shortcut to create a branch and check it out in one go is to use the `-b` flag, for example:

```
[user@host repository]$ git checkout -b wizard-dialog-prompt
Switched to a new branch 'wizard-dialog-prompt'
```

This will simultaneously create a branch named `wizard-dialog-prompt` and set that branch as your current location.

When running `git commit`, it is important to know which branch you have checked out, because this is the *only* branch that gets updated to have the new commit.

**Note**

Git tracks your current branch by maintaining a special pointer named **HEAD**. For example, when you run `git checkout main`, Git *attaches* the **HEAD** pointer to the **main** branch.

When you create a new commit, Git sets the parent of the new commit to the commit in which **HEAD** ultimately points. If **HEAD** points to a branch, that branch is updated to point to the new commit.

If you have ever seen Git report **detached HEAD state**, that is simply when **HEAD** does not point to a branch, but directly to a commit. Committing in this state can make it difficult to traverse back to newly created commits. There are several ways to get into this state, but the easiest way to get out of it is to use `git checkout` to attach **HEAD** to a branch.

Merging

Once a branch is considered done, then its changes are ready to be incorporated back into the main branch. This is called *merging* and is as equally important as branching.

If a branch creates a *divergence* in history, then merging *converges* it back.

The Git command for merging is `git merge`. It accepts a *single* argument for the branch to be merged *from*. Implicitly, your currently checked out branch is the branch that is merged into.

For example, assume you are on a branch named **simple-change**. Running `git merge other-change` would merge the commits from **other-change** *into* **simple-change**. In this case, **simple-change** gets updated, but **other-change** does not.

When running `git merge`, there are three possible outcomes:

- A clean *fast-forward merge*
- A clean merge with a *merge commit*
- A merge with *conflicts*

Fast-forward Merges

A fast-forward merge happens when the branch being merged into is a direct ancestor of the branch being merged from.

Git simply moves the older branch to point to the same commit as the newer branch. This is where the name comes from: it is as though the older branch is fast-forwarding through time.

For example, you have the following structure in your repository:

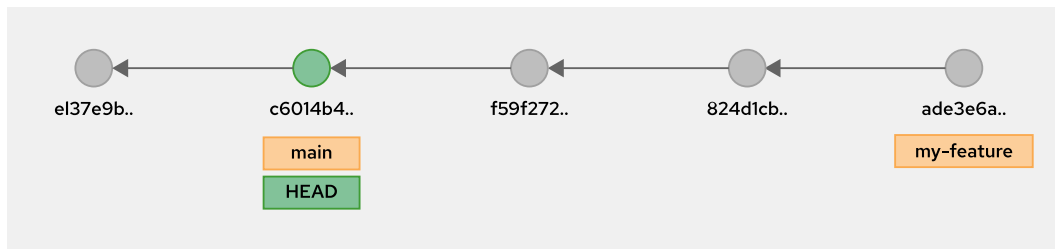


Figure 2.3: Example repository structure

**Note**

Remember that HEAD indicates that you have the `main` branch checked out.

Running `git merge my-feature` would result in the following structure:

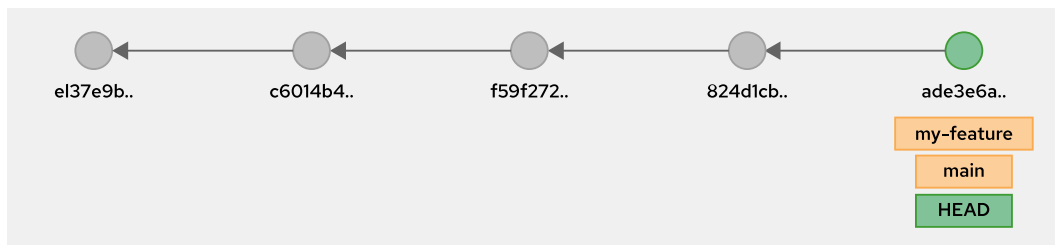


Figure 2.4: Example fast-forward merge

Because the two branches did not diverge, Git was able to simply move `main` to `ade3e6a`. The `my-feature` branch did not change.

Merges with Merge Commits

When you perform a merge, Git tries to automatically incorporate the changes made on both branches. If the branches have diverged, then Git must create a merge commit. *Merge commits* are a special type of commit that store the calculated changes resulting from the merge.

This time, your repository displays as follows:

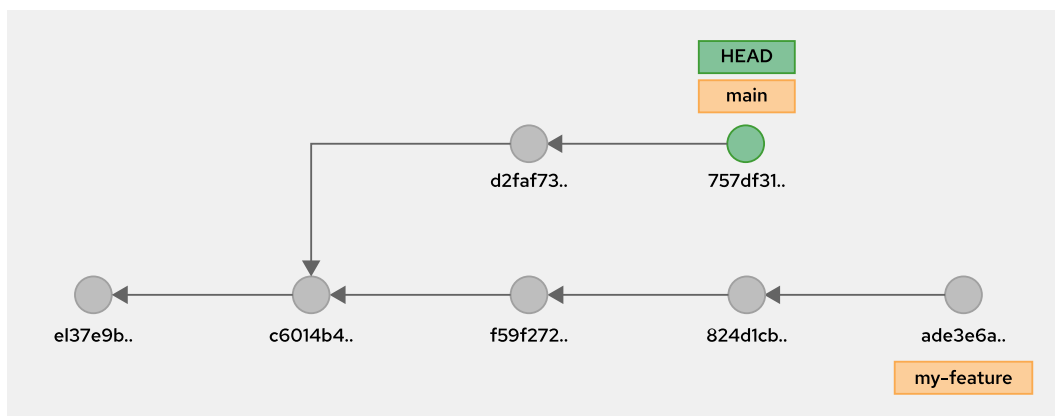


Figure 2.5: Example diverging repository structure

Note that `main` and `my-feature` have *diverged*. The `main` branch is no longer an ancestor of the `my-feature` branch.

Running `git merge my-feature` would result in the following structure:

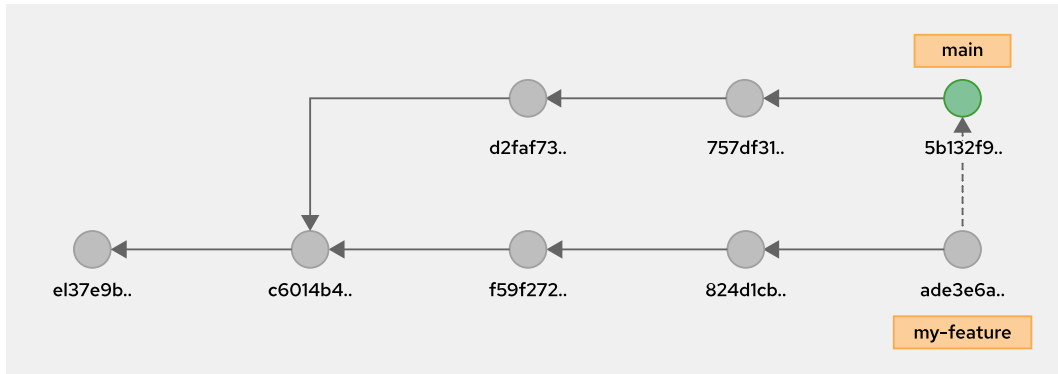


Figure 2.6: Example merge resulting in a merge commit

In this case, the `main` branch still moved, but a new merge commit with the hash `5b132f9` was created. Also note that the `my-feature` branch was *not* updated.



Note

Another way to remember merge direction is that Git only updates the branch to which HEAD is attached. Metaphorically, it is as though the branch is *attached* to HEAD via a rope or chain, and thus gets updated when HEAD does.

Merges with Conflicts

When performing a merge, a *conflict* occurs under the following criteria:

- The branches have diverged.
- Both branches include changes to the *same part* of the *same file*.

In these instances, Git does not have a way to automatically merge the changes. It cannot know which version is correct.

For every conflict that Git detects, it modifies the file to include both changes. It wraps each option in *conflict markers* and puts the local repository into a conflict resolution state.

Conflict markers appear in the files as sequences of `<`, `>`, and `=`, as shown in the following example:

```

<<<<<<< HEAD
Hi there!
=====
Hello there!
>>>>>>> your-feature-branch
  
```

In this state, the only valid commands are to fix the conflicts or abort the merge with `git merge --abort`.

To continue the merge, open the files and fix them. Be sure to remove all conflict markers. Then, stage the files by using `git add`. Finally, complete the merge by using `git commit`.

At this point, the resulting merge commit and overall structure are no different from a divergent merge without conflicts.

In the corresponding Guided Exercise, you will practice fixing a conflict.



References

An update on Red Hat's conscious language efforts

<https://www.redhat.com/en/blog/update-red-hats-conscious-language-efforts>

Git Manual

<https://www.man7.org/linux/man-pages/man1/git.1.html>

Git Pro – Basic Branching and Merging

<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

► Guided Exercise

Creating Git Branches

In this exercise, you will create a feature branch and track new changes on that branch.

Outcomes

You should be able to create branches, add commits to those branches, and switch between branches.

Before You Begin

To perform this exercise, ensure you have Git installed on your computer and access to a command line terminal.



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start every guided exercise from this folder.

Instructions

- 1. Initialize a new repository.
 - 1.1. Create a new folder named `hello-branches`. This folder will contain your repository.

```
[user@host D0400]$ mkdir hello-branches
```

- 1.2. Navigate to the `hello-branches` folder and initialize a new repository.

```
[user@host D0400]$ cd hello-branches
[user@host hello-branches]$ git init .
Initialized empty Git repository in ...output omitted.../D0400/hello-branches/.git/
```

- 1.3. Create the `helloworld.py` file with the following content:

```
def say_hello(name):
    print(f"Hello, {name}!")

say_hello("world")
```

- 1.4. Run the program:

```
[user@host hello-branches]$ python helloworld.py
Hello, world!
```


**Important**

In some Python installations, the Python binary is `python3` instead of `python`.

- 1.5. Stage and commit the file:

```
[user@host hello-branches]$ git add helloworld.py
[user@host hello-branches]$ git commit -m "added helloworld"
[master (root-commit) d4e8453] added helloworld
1 file changed, 4 insertions(+)
create mode 100644 helloworld.py
```

The `-m` flag is a shortcut used to set the commit message. Without it, the default editor opens in order to prompt for the message.

**Note**

`d4e8453` refers to the "commit hash" of the commit that was just created. It will differ on your machine.

- 1.6. Rename the current branch `master` to `main`

```
[user@host hello-branches]$ git branch -M main
```

- ▶ 2. Create a new feature branch.

- 2.1. Create a new branch named `different-name`.

```
[user@host hello-branches]$ git checkout -b different-name
Switched to a new branch 'different-name'
```

The `-b` flag is a shortcut to create the branch before switching to it.

- 2.2. Check that your current branch is the newly created `different-name`.

```
[user@host hello-branches]$ git status
On branch different-name
nothing to commit, working tree clean
```

- ▶ 3. Make and commit changes to the `helloworld.py` file.

- 3.1. Update the `helloworld.py` file to match:

```
def say_hello(name):
    print(f"Hello, {name}!")

say_hello("Pablo")
```

- 3.2. View the changes made to `helloworld.py`.

```
[user@host hello-branches]$ git diff
diff --git a/helloworld.py b/helloworld.py
index 3c29293..f69eef6 100644
--- a/helloworld.py
+++ b/helloworld.py
@@ -1,4 +1,4 @@
  def say_hello(name):
      print(f"Hello, {name}!")

-say_hello("world")
+say_hello("Pablo")
```

**Note**

Git uses a pager when the changes to display are too large to fit the screen. In this pager you can scroll up and down to see the changes. Press `q` to exit the pager.

3.3. Stage and commit the changes to `helloworld.py`:

```
[user@host hello-branches]$ git add .
[user@host hello-branches]$ git commit -m 'change name'
[different-name daf5db4] change name
1 file changed, 1 insertion(+), 1 deletion(-)
```

▶ 4. Compare the commit history between the `main` and `different-name` branches.4.1. View the commit history. The `different-name` branch has one more commit than the `main` branch.

```
[user@host hello-branches]$ git log
commit daf5db416ffd47ab6d95a16eaaaf4dabbcfa2cb72 (HEAD -> different-name)
Author: Your User Name <your.email@example.com>
Date:   Mon Sep 28 16:42:04 2020 -0400

    change name

commit d4e8453f6bc58a757a15f5ace664b3cd9afb65f6 (main)
Author: Your User Name <your.email@example.com>
Date:   Mon Sep 28 16:32:52 2020 -0400

    added helloworld
```

**Note**

The Git log also uses a pager when the commit log is too large to fit the screen. Press `q` to exit the pager.

▶ 5. Compare and merge the `my-feature-branch` branch.5.1. Switch to the `main` branch:

```
[user@host hello-branches]$ git checkout main
Switched to branch 'main'
```

5.2. View the commit history of the `main` branch:

```
[user@host hello-branches]$ git log
commit d4e8453f6bc58a757a15f5ace664b3cd9afb65f6 (HEAD -> main)
Author: Your User Name <your.email@example.com>
Date:   Mon Sep 28 16:32:52 2020 -0400

    added helloworld
```

There is one less commit than what was on the `different-name` branch.

5.3. Open `helloworld.py` in your editor. The file is missing the changes that were made on the `different-name` branch. It should have these contents:

```
def say_hello(name):
    print(f"Hello, {name}!")

say_hello("world")
```

5.4. Merge the `different-name` branch into the `main` branch. This performs a "fast-forward" merge. A fast-forward merge does not create a "merge commit". Instead, the merge requires only the branch itself to be moved to a different commit.

```
[user@host hello-branches]$ git merge different-name
Updating d4e8453..daf5db4
Fast-forward
 helloworld.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

5.5. View the updated commit history of the `main` branch:

```
[user@host hello-branches]$ git log
commit daf5db416ffd47ab6d95a16eaf4dabbcfa2cb72 (HEAD -> main, different-name)
Author: Your User Name <your.email@example.com>
Date:   Mon Sep 28 16:42:04 2020 -0400

    change name

commit d4e8453f6bc58a757a15f5ace664b3cd9afb65f6
Author: Your User Name <your.email@example.com>
Date:   Mon Sep 28 16:32:52 2020 -0400

    added helloworld
```

5.6. Delete the `different-name` branch

```
[user@host hello-branches]$ git branch -d different-name
Deleted branch different-name (was daf5db4).
```

The `-d` flag deletes the branch.

► **6.** Create a new branch and commit changes.

6.1. Create and check out a branch named `goodbye-name`:

```
[user@host hello-branches]$ git checkout -b goodbye-name
Switched to a new branch 'goodbye-name'
```

6.2. Open `helloworld.py` in your editor and change `Hello` to `Goodbye`:

```
def say_hello(name):
    print(f"Goodbye, {name}!")

say_hello("Pablo")
```

6.3. Stage and commit the changes:

```
[user@host hello-branches]$ git commit -a -m 'say goodbye'
[goodbye-name 9e43ceb] say goodbye
1 file changed, 1 insertion(+), 1 deletion(-)
```

Including the `-a` flag stages *all* local changes before creating the commit.

► **7.** Make a conflicting commit on the `main` branch.

7.1. Run `git checkout main` Switch back to the `main` branch:

```
[user@host hello-branches]$ git checkout main
Switched to branch 'main'
```

7.2. Open `helloworld.py` in your editor and change `Hello` to `Welcome`:

```
def say_hello(name):
    print(f>Welcome, {name}!")

say_hello("Pablo")
```

7.3. Stage and commit the changes:

```
[user@host hello-branches]$ git commit -a -m 'say welcome'
[main c80d322] say welcome
1 file changed, 1 insertion(+), 1 deletion(-)
```



Note

Good practice suggests not committing changes directly to the `main` branch. It is better to create a "feature branch" to house the changes. It is done here for the sake of example.

► **8.** Compare and merge the `goodbye-name` branch.

- 8.1. View the differences between the contents of `helloworld.py` on the `main` and `goodbye-name` branches:

```
[user@host hello-branches]$ git diff goodbye-name
diff --git a/helloworld.py b/helloworld.py
index 4e8dd88..22604ac 100644
--- a/helloworld.py
+++ b/helloworld.py
@@ -1,4 +1,4 @@
 def say_hello(name):
-    print(f"Goodbye, {name}!")
+    print(f"Welcome, {name}!")

say_hello("Pablo")
```

- 8.2. Merge the `goodbye-name` branch into the `main` branch.

```
[user@host hello-branches]$ git merge goodbye-name
Auto-merging helloworld.py
CONFLICT (content): Merge conflict in helloworld.py
Automatic merge failed; fix conflicts and then commit the result.
```

A conflict occurred while performing the merge. This is due to `main` and `goodbye-name` incorporating changes to the same line in the same file since the branches diverged.

- 8.3. Open `helloworld.py` in your editor. This is how the file will first appear:

```
def say_hello(name):
<<<<<< HEAD
    print(f"Welcome, {name}!")
=====
    print(f"Goodbye, {name}!")
>>>>>> goodbye-name

say_hello("Pablo")
```

Git indicates the conflicting lines by using "conflict markers". These are lines beginning with sequences of `<`, `>`, and `=`. Specifically, the first block contains the changes from the `main` branch (`HEAD`), and the second block contains changes from the `goodbye-name` branch.

- 8.4. Fix the conflict by removing the conflict markers and correcting the code to match:

```
def say_hello(name):
    print(f"Goodbye, {name}!")

say_hello("Pablo")
```

**Note**

After fixing conflicts, any source code should be fully functional and pass any tests. For example, if your programming language has a compilation step, the code should successfully compile.

Be cautious not to accidentally commit any conflict markers into your repository.

8.5. Run the program to ensure it is still working:

```
[user@host hello-branches]$ python helloworld.py
Goodbye, Pablo!
```

8.6. In order to indicate to Git that you have resolved the conflicts, stage and commit the conflicting file:

```
[user@host hello-branches]$ git commit -a
...output omitted...
[main 32d7b8c] Merge branch 'goodbye-name' into main
```

When a commit message is omitted upon committing to resolve a conflicting merge, Git will open the default editor with a default commit message.

8.7. Remove the goodbye-name branch:

```
[user@host hello-branches]$ git branch -d goodbye-name
Deleted branch goodbye-name (was 9e43ceb).
```

► 9. View the commit history to see that main has all of the changes from both branches:

```
[user@host hello-branches]$ git log
commit 32d7b8c9b28c9b41cd5f4a97e63e8cb284622c1e (HEAD -> main)
Merge: c80d322 9e43ceb
Author: Your User Name <your.email@example.com>
Date: Tue Sep 29 18:45:02 2020 -0400

    Merge branch 'goodbye-name' into main

commit c80d32280f9b1165c9d74303b137c0bb0a8c59b5
Author: Your User Name <your.email@example.com>
Date: Tue Sep 29 17:53:45 2020 -0400

    say welcome

commit 9e43ceb14b042fee5c08d41a2130a075b1c74113
Author: Your User Name <your.email@example.com>
Date: Tue Sep 29 17:49:31 2020 -0400

    say goodbye

commit daf5db416ffd47ab6d95a16eaa4dabbcfa2cb72
Author: Your User Name <your.email@example.com>
```

```
Date:   Mon Sep 28 16:42:04 2020  -0400

    change name

commit d4e8453f6bc58a757a15f5ace664b3cd9afb65f6
Author: Your User Name <your.email@example.com>
Date:   Mon Sep 28 16:32:52 2020  -0400

    added helloworld
```

This concludes the guided exercise.

Managing Remote Repositories

Objectives

After completing this section, you should be able to manage and collaborate with remote repositories.

Explaining Remotes

As seen in *Building Applications with Git*, Git is a distributed and decentralized version control system. With Git, you can have multiple copies of the same repository, without a central server that hosts a primary copy.

Although Git does not force you to use a principal repository copy, it does allow you to do so. In fact, it is very common for teams to work with a primary repository, which centralizes the development and integrates the changes made by different contributors.

When not using a principal repository copy, developers create local repository copies in their workstations where they carry out their work. In contrast, the primary repository copy is normally hosted externally, in a dedicated server or a Git forge, such as GitHub [<https://github.com/>]. Because the primary repository copy is often remote, this copy is referred to as the *remote repository*, or simply as the **remote**.



Note

From the perspective of a local repository, a *remote* is simply a reference to another repository. You can add multiple *remotes* to a local repository to push changes to different remote repositories.

Although a *remote* often points to an external repository it does not necessarily have to do so. You can add a *remote* that points to another local repository in your workstation.

Remote Branches

When working with remote repositories, you can have both remote and local branches. If you work on a local repository connected with a remote repository, then you will have the following branch categories:

- **Local branches:** Branches that exist in your local repository. You normally organize, carry out your work, and commit changes by using local branches. For example, if you use the `git branch` command to create a new branch called `my-feature`, then `my-feature` is a local branch.
- **Remote branches:** Branches that exist in a remote repository. For example, all the branches in a GitHub repository are considered remote branches. Teams use remote branches for coordination, review and collaboration.
- **Remote-tracking branches:** Local branches that reference remote branches. Remote-tracking branches allow your local repository to push to the remote and fetch from the remote. Git names these branches by using the following convention: `remote_name/branch_name`.

For example, `origin/main` is a remote-tracking branch, because it tracks the `main` branch from the `origin` remote.

Git uses these remote-tracking branches to coordinate with remote Git servers when synchronizing branches. You can also use remote-tracking branches to keep track of remote branches.

To download changes from the remote, you first **fetch** the remote changes into a remote-tracking branch and then merge it into the local branch. As you will see later in this section, you can also **pull** changes to update the local branch. To upload changes to the remote, you must **push** the changes from your local branch to a remote-tracking branch for Git to update the remote.

Working with Remotes

After creating the remote repository, you can add the remote repository as a new *remote* to the local repository. You can use any Git forge or your own infrastructure to create, manage, and serve remote repositories. This course covers GitHub, a well-known Git forge.

Creating a New GitHub Repository

GitHub, like other Git forges, offers a full-fledged interface to manage Git repositories, improve DevOps processes, and relieve users from the hassle of having to maintain their own Git server infrastructure. The steps to create a new repository in GitHub are the following:

1. As an authenticated user in GitHub, click + in the navigation bar, and then click **New repository**.
2. In the **Create a new repository** page, enter a value for the **Repository name** field. You can optionally add a description and initialize the repository with some files, such as a *README* file, a *.gitignore* file or a *LICENSE* file.

Note

It is good practice to add a description, as well as the following files to your GitHub repositories:

- A *README* file, which describes and explains your project.
- A *.gitignore* file, which indicates to Git what files or folders to ignore in a project.
- A *LICENSE* file, which defines the current license of your project.

Figure 2.7: GitHub new repository form

After you have created the remote repository in GitHub, you can start adding code. To start working in the new repository from your local workstation, you generally have two options:

- Clone the remote repository as a new local repository in your workstation.
- Add the remote repository as a *remote* to a local repository in your workstation.

Cloning a Repository

When you clone a remote repository, Git downloads a copy of the remote repository and **creates a new repository in your local machine**. This newly created repository is the **local repository**. To clone a repository, use the `git clone` command, specifying the URL of the repository that you want to clone.

```
[user@host ~]$ git clone https://github.com/YOUR_GITHUB_USER/your-repo.git
Cloning into 'your-repo'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 12 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (12/12), done.
```



Note

Git accepts an optional parameter for the local repository folder: `git clone repository-url DESTINATION_FOLDER`. If you do not specify this parameter, then git creates the local repository in a folder with the same name as the remote repository.

Adding a Remote to a Local Repository

As a part of the clone process, Git configures the local repository with a *remote* called *origin*, which points to the remote repository. After cloning the repository, you can navigate to the newly created local repository folder, and check that Git has configured a *remote* that points to the remote repository. Use the `git remote -v` command for this.

```
[user@host ~]$ cd your-repo
[user@host your-repo]$ git remote -v
origin https://github.com/your_github_user/your-repo.git (fetch)
origin https://github.com/your_github_user/your-repo.git (push)
```

In order to contribute to a remote repository, you must have a remote set up that points to that repository. As you saw in the preceding example, using `git clone` provides the `origin` remote automatically. You can also add **additional remotes**.

When you add a new remote to an existing local repository, you must identify the remote by choosing a **remote name**, as well as specify the **URL of the remote repository**.

**Note**

Although you can name your remotes anything you like, the standard is to use `origin` as the name for the first remote added to a local repository. Additional remotes should use different names, because remote names must be unique.

To add a new remote, use the `git remote add` command, followed by a name and the URL of the remote repository.

```
[user@host your-local-repo]$ git remote add \
my-remote https://github.com/YOUR_GITHUB_USER/your-repo.git
[user@host your-local-repo]$
```

Notice how the new remote is called `my-remote` and points to `https://github.com/your_github_user/your-repo.git`. Having additional remotes is common when you contribute to third-party remote repositories, by using *forks*. With multiple *remotes* in a local repository, you can decide which remote to pull from and push to.

**Note**

A *fork* is just a copy of a remote repository used to securely contribute to third-party repositories. You will learn more about forks in subsequent sections.

Pushing to the Remote

With a remote associated to your local repository, you are now ready to push your local commits to the remote repository. When you create new commits in your local branch, the remote-tracking branch and the remote branch become outdated with respect to the local branch, as shown in the following figure:

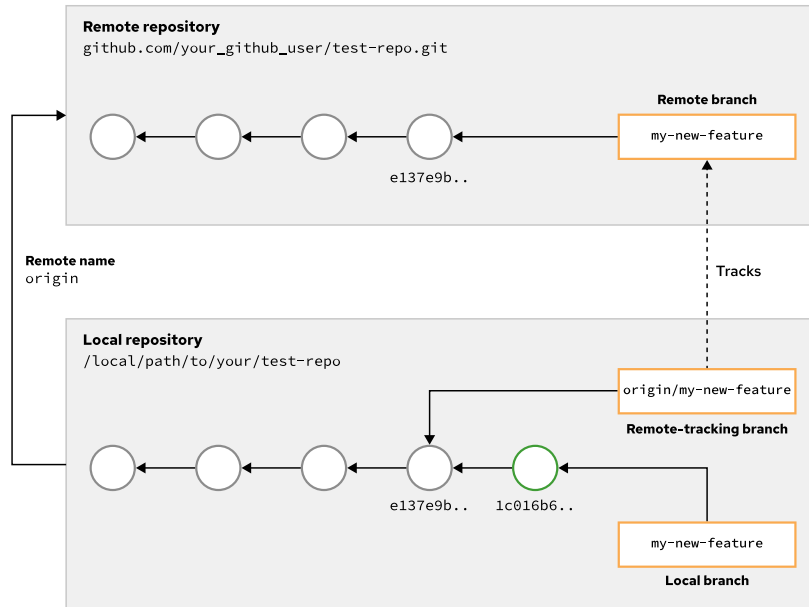


Figure 2.8: The local my-new-feature branch is ahead of the origin/my-new-feature branch and the remote my-new-feature branch

The figure shows how the new `1c016b6..` commit is in the local branch only. To update the remote branch with your new commit, you must push your changes to the remote. To push changes, use the `git push` command, specifying the *remote* and the branch that you want to push. When using the `git push` command, you push the currently selected local branch.

```
[user@host your-local-repo]$ git status
On branch my-new-feature ❶
...output omitted...
[user@host your-local-repo]$ git push origin my-new-feature ❷
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 297 bytes | 297.00 KiB/s, done.
Total 3 (delta 0), reused 1 (delta 0)
To https://github.com/your_github_user/test-repo.git
e137e9b..1c016b6 my-new-feature -> my-new-feature
```

The `git push` command pushes the currently selected `my-new-feature` local branch to the remote `my-new-feature` branch in GitHub and consequently updates the remote-tracking `origin/my-new-feature` branch.

- ❶ Check that the currently selected local branch is `my-new-feature`.
- ❷ Push the local `my-new-feature` branch to the GitHub `my-new-feature` branch and update the remote-tracking `origin/my-new-feature` branch.

**Important**

If the remote branch does not exist when you run the `git push` command, then Git creates both the remote branch and the remote-tracking branch.

After the push, the local, remote, and remote-tracking branches all point to the same commit. At this point, the remote branch is synchronized with the local branch.

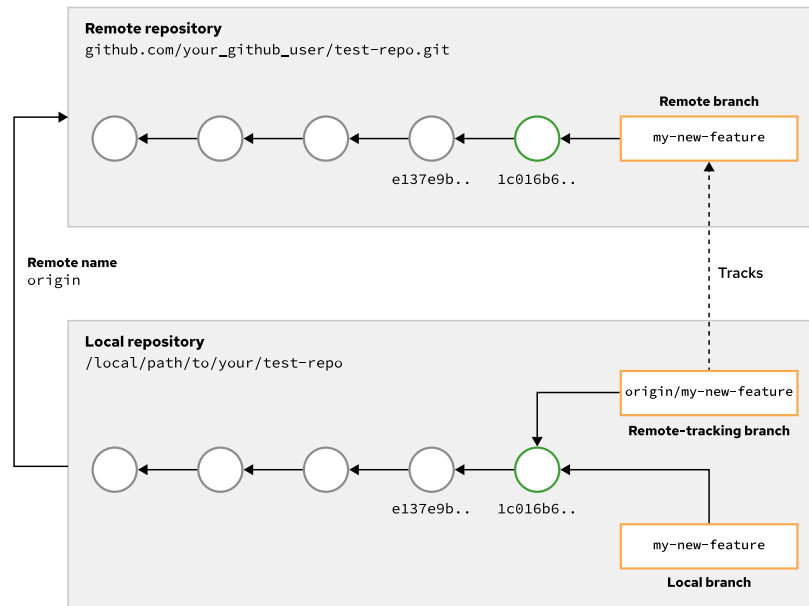


Figure 2.9: The remote branch is up to date with the local branch

**Note**

You might have seen or used `git push` without providing the remote or branch name arguments. This is achieved by setting a *branch upstream* with the `--set-upstream` or simply `-u` option when pushing a new branch for the first time. Be careful not to confuse it with the concept of an upstream *fork* or a *remote* named upstream.

Pull Requests

When you finish your work in a branch, normally you would want to propose your work for integration. To propose changes to the main development branch in the GitHub remote, you can either just merge and push the changes, or instead use *pull requests*.

If you just merge your work into the main branch and push it to the remote, then your team does not have a chance to review your changes. In contrast, if you use a pull request, then your code goes through manual and automatic validations, which improve the quality of your work.

Pull requests are a powerful web-based feature included in most Git forges, including GitHub. Pull requests allow teams to review and validate proposed changes before their integration into the main branch. Pull requests help achieve the following:

- Compare branches

- Review proposed changes
- Add comments
- Reject or accept changes

Pull requests also provide a clear point to run validation and testing checks via *hooks*, which you will see in a later chapter.

The steps to create a pull request in GitHub are as follows:

1. In your GitHub repository main page, click **Pull requests** to navigate to the pull requests page. In the pull requests page, click **New pull request**.
2. In the branches selection area, click the **base: main** selector to select the branch that you want to integrate your changes into. Click **compare: branch** to select the branch containing the changes that you want to merge into the *base* branch. These two branches will be eventually merged, after the team agrees on accepting the pull request.

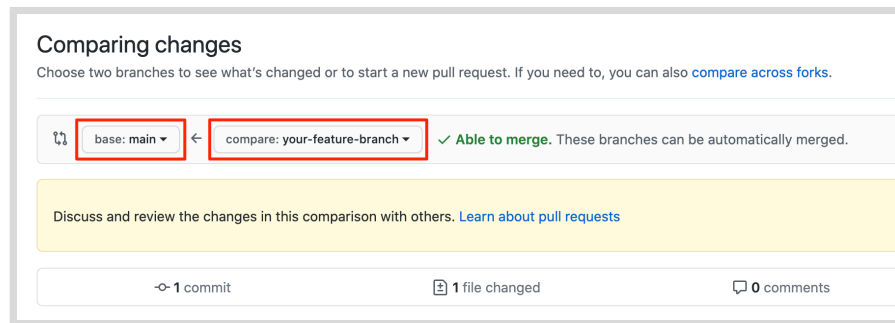


Figure 2.10: GitHub view to compare changes

After you select the branches, GitHub automatically shows the changes to be integrated from the *compare* branch into the *base* branch.

3. Click the **Create pull request** button to open the pull request creation form. The form automatically suggests a title for the pull request, which you can change. You can also add a description to the pull request. Also, use this form to request a review from other users, assign the pull request to someone, set labels, projects, and milestones.

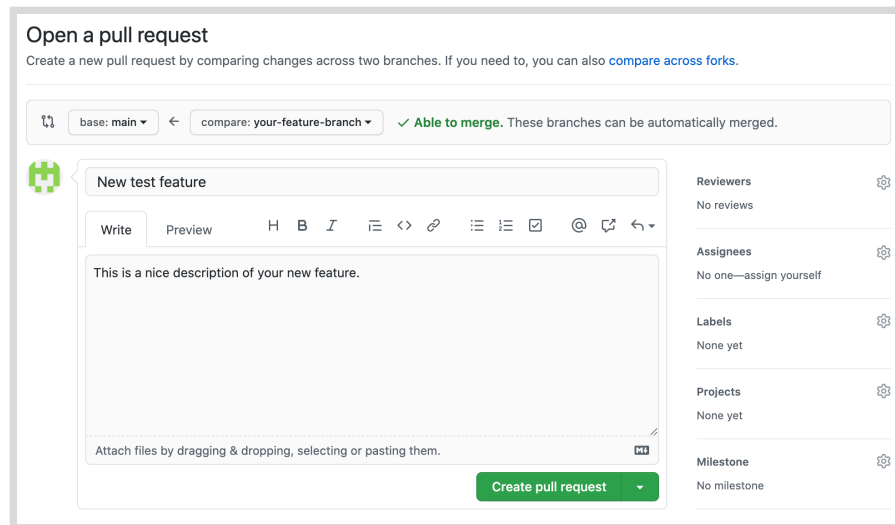


Figure 2.11: GitHub pull request form

**Note**

Use the pull request description field to describe what your changes are about.

For example, if the pull request fixes an error, the description usually includes a description of the problem. If the pull request includes a feature, the description field describes the new feature.

The description is also a good place to include informative comments about intent, clarifications, or warnings.

4. Finally, click **Create pull request** to create and submit the pull request. By default, GitHub will send a notification to the assignees and reviewers.

**Note**

GitHub users can modify notification settings. Users that choose not to be notified will not receive pull request notifications.

After you create a pull request, your team validates it. The validation process comprises code reviews and automatic validation via Continuous Integration pipelines. The pull request is ready to be merged after the pull request is reviewed and validated.

When you are ready to merge the pull request, click **Merge pull request** in the pull request page.

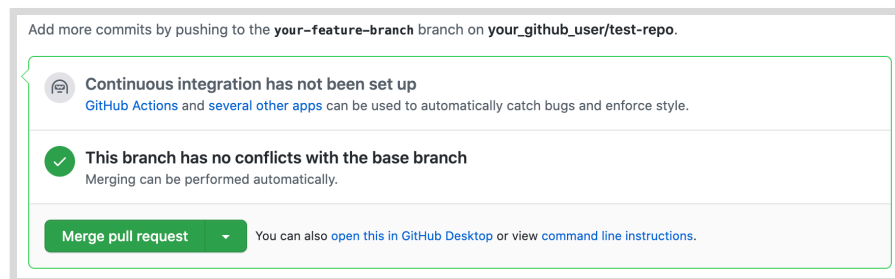


Figure 2.12: GitHub pull request merge pane

Merging a pull request triggers the git merge process underneath. From a Git perspective, it produces the same effect as if you ran the `git merge` command locally.

Pulling from the Remote

As you work in your local repository, chances are that other team members contribute and push changes to the remote repository. To keep your local copy up to date with the remote, you must pull changes from the remote frequently.

Fetching and Merging Changes

Pretend that a team member merged a pull request and integrated new changes into the `main` branch of your repository. Now your local `origin/main` remote-tracking branch is one commit behind the remote, as shown in the following figure. The local `main` branch is also behind the remote because it points to the same commit as the `origin/main` branch.

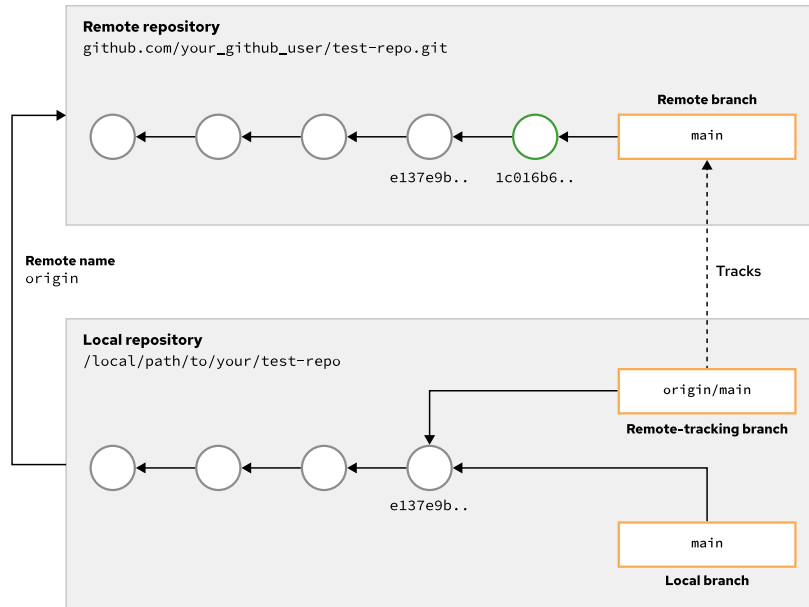


Figure 2.13: The local main and origin/main branches are behind the remote main branch

To retrieve the latest commits from the remote repository and update remote-tracking branches, use `git fetch`.

```
[user@host your-local-repo]$ git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/your_github_user/test-repo
e137e9b..1c016b6 main -> origin/main
```

The `git fetch` command updates your local repository by downloading branches, commits, and other Git references and objects. By default, this command **only updates remote-tracking branches**.



Note

The `git fetch` command optionally accepts the name of the remote as a parameter. If this parameter is not specified, the `git fetch` command uses `origin` as the remote name by default. Therefore, if your local repository has a remote called `origin`, calling `git fetch origin` is equivalent to calling `git fetch`.

After fetching changes from the remote, you have updated the remote-tracking `origin/main` branch. However, your local `main` branch is still behind, as the following figure shows.

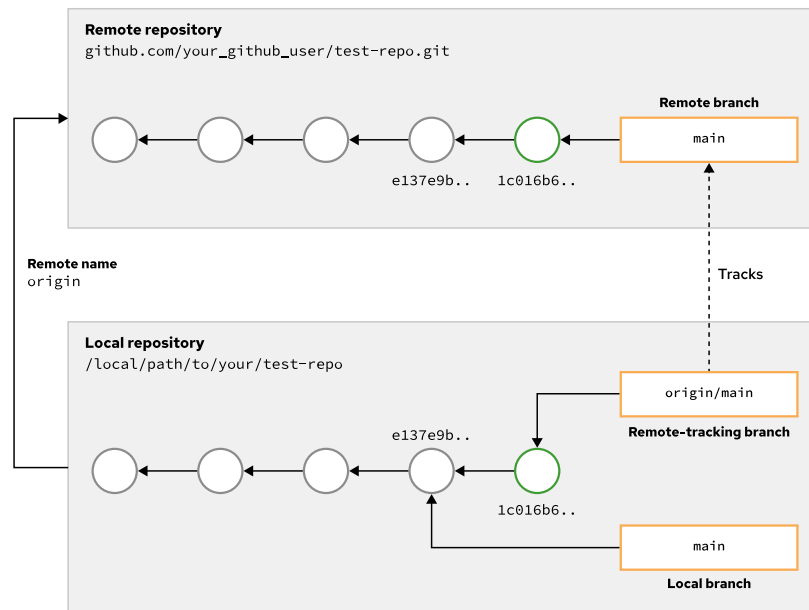


Figure 2.14: The local main branch is behind the remote-tracking origin/main branch

To also update your local main branch, you must merge the origin/main remote-tracking branch into your local main branch, by using the `git merge` command.

```
[user@host your-local-repo]$ git checkout main
Switched to branch 'main'
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)
[user@host your-local-repo]$ git merge origin/main
Updating e137e9b..1c016b6
Fast-forward
...output omitted...
```

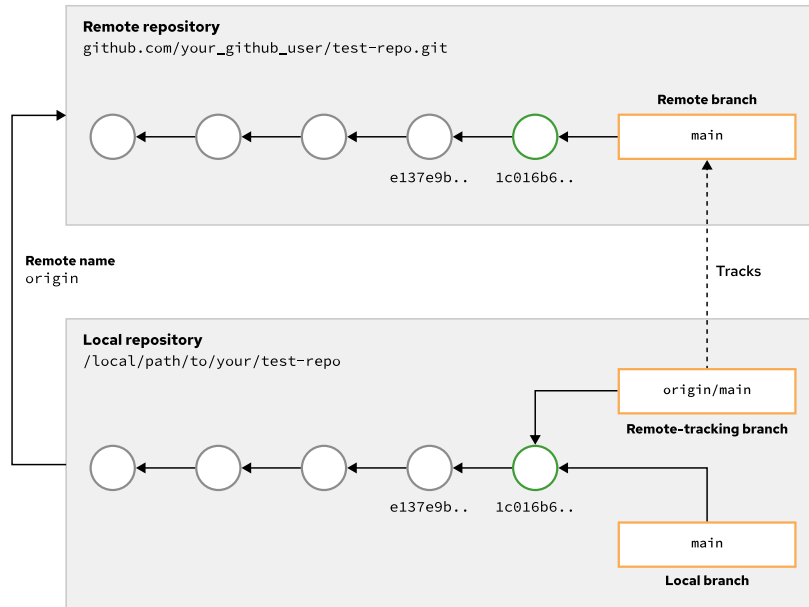


Figure 2.15: The local main branch is up to date with the remote main branch

After the merge, the local branch is up to date with the remote.



Important

Note that the preceding example shows a simple merge scenario. If you have added commits directly to your local `main` branch, then Git will add a *merge commit*. You must then push this commit to the remote. You and your teammates might need to fix conflicts for both of these steps.

It is better to avoid this scenario by not directly committing to the `main` branch, even locally.

Pulling Changes

As an alternative to running `fetch` and `merge` manually, you can use the `git pull` command to update your local branch in one command. Similar to the `git push` command, you must specify the name of the *remote* and the name of the remote branch. The `git pull` command applies the changes to the currently selected local branch.

```
[user@host your-local-repo]$ git checkout main ❶
Already on 'main'
Your branch is up to date with 'origin/main'.
[user@host your-local-repo]$ git pull origin main ❷
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/your_github_user/test-repo
   e137e9b..1c016b6  main      -> origin/main
Updating e137e9b..1c016b6
```

Fast-forward

```
README.md | 1 +  
1 file changed, 1 insertion(+)
```

- 1 Select the local branch that you want to update.
- 2 Pull from a specific *remote* and branch. The command fetches changes to the remote-tracking branch and merges the remote-tracking branch into the local branch.



Note

If your local branch is already associated with a remote-tracking *upstream* branch, then you can just run `git pull`.



References

Git Basics - Working with Remotes

<https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes>

Git Branching - Remote Branches

<https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches>

About pull requests

<https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests>

► Guided Exercise

Managing Remote Repositories

In this exercise, you will learn how to use and manage remote Git repositories and how to review changes by using pull requests.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to set up a remote repository, push code to and pull code from the remote repository and use pull requests to review changes.

Before You Begin

To perform this exercise, ensure you have Git installed on your computer and access to a command line terminal.



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start every guided exercise from this folder.

Instructions

- 1. Initialize a local repository.

- 1.1. Create a new folder named `hello-remote`. This folder will contain your repository.

```
[user@host D0400]$ mkdir hello-remote
```

- 1.2. Navigate to the `hello-remote` folder and initialize a new repository:

```
[user@host D0400]$ cd hello-remote
[user@host hello-remote]$ git init .
Initialized empty Git repository in ...output omitted.../D0400/hello-remote/.git/
```

- 1.3. Create the `helloworld.py` file with the following content:

```
print("Hello world!")
```

- 1.4. Stage and commit the file:

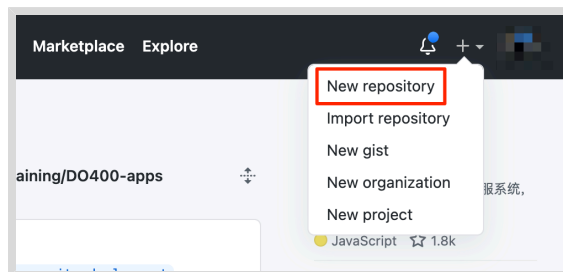
```
[user@host hello-remote]$ git add helloworld.py
[user@host hello-remote]$ git commit -m "added helloworld"
```

- 1.5. Rename the current branch `master` to `main`:

```
[user@host hello-remote]$ git branch -M main
```

► 2. Create a new repository in GitHub.

- 2.1. Open a web browser and navigate to `https://github.com`.
- 2.2. Sign in with your GitHub credentials. If you do not have an account, then create one.
- 2.3. Click + → **New repository** in the upper right of the window.



- 2.4. Fill in the **Repository name** field with the `hello-remote` value. Do not change any other field. Click **Create repository**.

 A screenshot of the GitHub 'Create new repository' form. The 'Owner' field is set to 'your_github_user' and the 'Repository name' field is set to 'hello-remote' with a green checkmark. Below these fields is a hint: 'Great repository names are short and memorable. Need inspiration? How about `furry-octo-fortnight`?'. There is an empty 'Description (optional)' text box. Under the 'Visibility' section, the 'Public' radio button is selected. The 'Initialize this repository with:' section has three unchecked checkboxes: 'Add a README file', 'Add .gitignore', and 'Choose a license'. At the bottom is a green 'Create repository' button.

► 3. Add the remote to the local repository.

- 3.1. Switch to the command line and check you are in the `hello-remote` folder.
- 3.2. Verify that no remotes exist:

```
[user@host hello-remote]$ git remote -v
[user@host hello-remote]$
```

3.3. Use the `git remote add` command to add the remote repository:

```
[user@host hello-remote]$ git remote add \
origin https://github.com/YOUR_GITHUB_USER/hello-remote.git
```

3.4. Verify that the remote has been added:

```
[user@host hello-remote]$ git remote -v
origin https://github.com/your_github_user/hello-remote.git (fetch)
origin https://github.com/your_github_user/hello-remote.git (push)
```

3.5. Set the upstream branch and push to the remote repository. Notice the use of the `--set-upstream` parameter to set the upstream branch. You must provide your username and personal access token to push changes to the GitHub remote repository.

```
[user@host hello-remote]$ git push --set-upstream origin main
Username for 'https://github.com': YOUR_GITHUB_USER
Password for 'https://
your_github_user@github.com': YOUR_GITHUB_PERSONAL_ACCESS_TOKEN
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 247 bytes | 247.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/your_github_user/hello-remote.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

3.6. In a web browser, navigate to `https://github.com/YOUR_GITHUB_USER/hello-remote`. Check your changes are visible in GitHub.

- ▶ 4. Create a feature branch to personalize the greeting message. The program must read your name as a command-line parameter and show it in the greeting message.

4.1. Create a new branch named `improve-greeting`:

```
[user@host hello-remote]$ git branch improve-greeting
```

4.2. Checkout the `improve-greeting` branch:

```
[user@host hello-remote]$ git checkout improve-greeting
Switched to branch 'improve-greeting'
```



Note

You can perform both the creation and the checkout in a single command by using the `git checkout -b improve-greeting` command.

4.3. Modify the `helloworld.py` file with the following contents:

```
import sys

print("Hello {}".format(sys.argv[1]))
```

4.4. Run the program:

```
[user@host hello-remote]$ python helloworld.py Student
Hello Student!
```



Warning

In some Python installations, the python binary is `python3` instead of `python`.

4.5. Use the `git status` command to check the status of the repository:

```
[user@host hello-remote]$ git status
On branch improve-greeting
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   helloworld.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Although Git detects the changes, you must stage them for commit.

4.6. Stage the changes by using the `git add` command. Next, run the `git status` command again to check that the changes have been staged for commit:

```
[user@host hello-remote]$ git add helloworld.py
[user@host hello-remote]$ git status
On branch improve-greeting
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   helloworld.py
```

4.7. Commit the changes:

```
[user@host hello-remote]$ git commit -m "added parameter to greeting message"
[improve-greeting 6b5013e] added parameter to greeting message
1 file changed, 3 insertions(+), 1 deletion(-)
```

4.8. Set the branch upstream and push the branch to GitHub:

```
[user@host hello-remote]$ git push --set-upstream origin improve-greeting
...output omitted...
Enumerating objects: 5, done.
```

```

Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 316 bytes | 316.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'improve-greeting' on GitHub by visiting:
remote:   https://github.com/your_github_user/hello-remote/pull/new/improve-
greeting
remote:
To https://github.com/your_github_user/hello-remote.git
* [new branch]      improve-greeting -> improve-greeting
Branch 'improve-greeting' set up to track remote branch 'improve-greeting' from
'origin'.

```

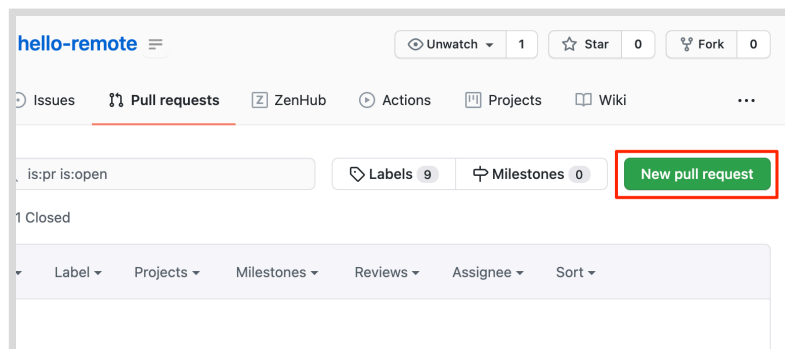
Note how the `improve-greeting` branch is created in the remote GitHub repository. Also notice how GitHub provides you with a URL to create a pull request for this new branch.



Note

Use the `--set-upstream` parameter when pushing a new local branch to the remote for the first time.

- 4.9. In a web browser, navigate to `https://github.com/YOUR_GITHUB_USER/hello-remote/tree/improve-greeting` to verify that the branch and the changes have been pushed to GitHub.
- ▶ 5. Review the changes introduced in the feature branch by using a pull request. After reviewing the code, merge the pull request to integrate the new feature into the `main` branch.
 - 5.1. In a web browser, navigate to `https://github.com/YOUR_GITHUB_USER/hello-remote`.
 - 5.2. Click **Pull requests** to navigate to the pull requests page. In the pull requests page, click **New pull request**.



- 5.3. In the branches selection area, click the **compare: main** selector to select the `improve-greeting` branch.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base: main
←
compare: improve-greeting
✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#)

Create pull request

1 commit
1 file changed
0 comments
1 contributor

Commits on Mar 11, 2021

added parameter to greeting message 747d2bb

Showing 1 changed file with 3 additions and 1 deletion.

Unified Split

4

helloworld.py

@@ -1,2 +1,4 @@

1 - print("Hello world!")

2

1 + import sys

2 +

3 + print("Hello {}".format(sys.argv[1]))

4

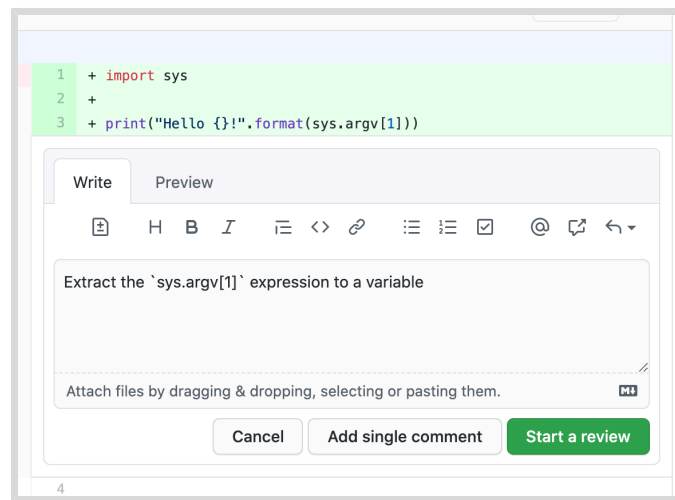
Next, click the **Create pull request** button to open the pull request creation form. In the pull request creation form, click **Create pull request**.

- 5.4. Click **Assignees** to assign the pull request to yourself. Click **Reviewers** to request reviews from other people.

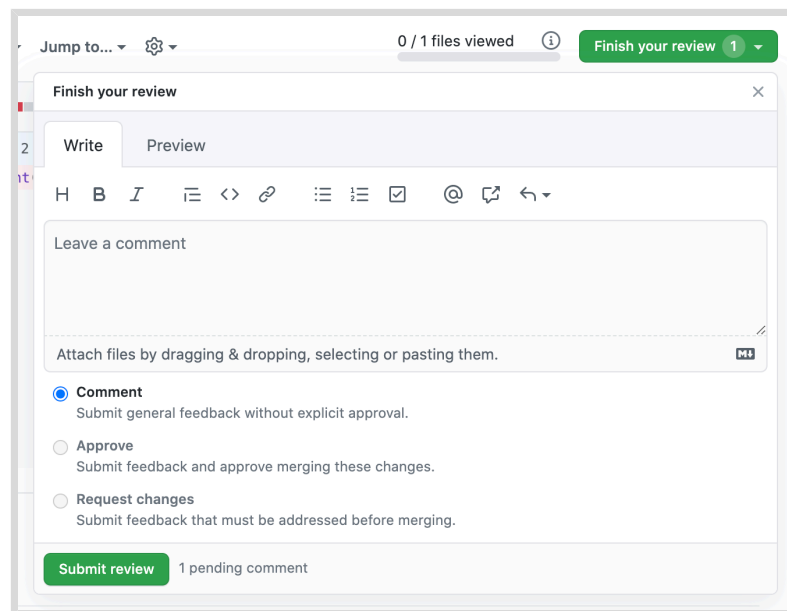
Note

Normally, you would assign the pull request to someone else for peer review.

- 5.5. Click the **Files changed** tab to observe the changes introduced in the `improve-greeting` branch, compared to the `main` branch.
- 5.6. Add a comment to suggest the extraction of the `sys.argv[1]` expression to a variable. Click the **+** button, which displays when you hover the mouse over the `print("Hello {}".format(sys.argv[1]))` line. Write the comment and click **Start a review**.



Click **Finish your review** → **Submit review** to finish the review with the comment.



- 6. Update the code to address the comment from the pull request and update the pull request.

6.1. Open the `helloworld.py` file and modify the code to extract `sys.argv[1]` into a variable. The file content should look like this:

```
import sys

name = sys.argv[1]
print("Hello {}".format(name))
```

6.2. Verify that the program runs successfully:

```
[user@host hello-remote]$ python helloworld.py Student
Hello Student!
```

- 6.3. Stage, commit and push the changes. The `-a` parameter adds the modified files to the stage before creating the commit.

```
[user@host hello-remote]$ git commit -a -m "extracted cli argument to variable"
[improve-greeting 2a54571] extracted cli argument to variable
1 file changed, 2 insertions(+), 1 deletion(-)
[user@host hello-remote]$ git push
...output omitted...
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 306 bytes | 306.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/your_github_user/hello-remote.git
6b5013e..2a54571 improve-greeting -> improve-greeting
```

- 6.4. Go back to GitHub, refresh the pull request page and verify that the changes show up. The previous comment should show up as **Outdated**.

► 7. Merge the pull request.



Note

You should not merge your own pull requests in all cases.

For example, if you contribute to a third party repository, then a project maintainer will instead merge the pull request.

- 7.1. In GitHub, navigate to the pull request page if you are not already there and click the **Merge pull request** button. Click **Confirm merge** on the confirmation form that shows up.

- 7.2. A **Delete branch** button is displayed immediately after the merge. Click this button to delete the `improve-greeting` remote branch.
- 7.3. In the local repository, checkout the `main` branch:

```
[user@host hello-remote]$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

Note the checkout message. The local repository is not aware of the remote changes.

- 7.4. Check that the changes you just merged are not in the `main` remote tracking branch.

```
[user@host hello-remote]$ git log origin/main
```

- 7.5. Fetch changes from the remote repository.

```
[user@host hello-remote]$ git fetch -p
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
Unpacking objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
From https://github.com/your_github_user/hello-remote
2a0444d..ada565e  main    -> origin/main
```

The `-p` parameter prunes remote-tracking branches that do not exist in the remote repository anymore, causing the removal of the `origin/improve-greeting` branch.

- 7.6. Check that the merged commits have been fetched. Run the `git log` command again:

```
[user@host hello-remote]$ git log origin/main
```

The log view should show the latest commits fetched from the remote `main` branch:

```
commit ...output omitted... (origin/main)
Merge: ...output omitted...
Author: ...output omitted...
Date:   Fri Sep 25 17:15:23 2020 +0200

    Merge pull request #1 from ...output omitted...

    added parameter to greeting message

commit ...output omitted... (origin/improve-greeting, improve-greeting)
Author: ...output omitted...
Date:   Fri Sep 25 17:11:18 2020 +0200

    extracted cli argument to variable

commit ...output omitted...
Author: ...output omitted...
Date:   Fri Sep 25 14:39:19 2020 +0200

    added parameter to greeting message

commit ...output omitted... (HEAD -> main)
Author: ...output omitted...
Date:   Fri Sep 25 13:37:44 2020 +0200

    added helloworld
```

If Git shows the output by using a pager, then press `q` to quit the log screen.

7.7. Run git log against the main branch.

```
[user@host hello-remote]$ git log
```

The log should only show one commit in the local main branch.

```
commit ...output omitted... (HEAD -> main)
Author: ...output omitted...
Date:   Fri Sep 25 13:37:44 2020 +0200

    added helloworld
```

If Git shows the output by using a pager, then press q to quit the log screen.

7.8. Merge the remote-tracking origin/main branch into the local main branch.

```
[user@host hello-remote]$ git merge origin/main
Updating 2a0444d..ada565e
Fast-forward
 helloworld.py | 5 +++-
 1 file changed, 4 insertions(+), 1 deletion(-)
```

7.9. Delete the local feature branch:

```
[user@host hello-remote]$ git branch -d improve-greeting
Deleted branch improve-greeting (was 2a54571).
```

7.10. Run git log to show the commits on the main branch. All the commits should be in this branch now.

```
commit ...output omitted... (HEAD -> main, origin/main)
Merge: ...output omitted...
Author: ...output omitted...
Date:   Fri Sep 25 17:15:23 2020 +0200

    Merge pull request #1 from ...output omitted...

    added parameter to greeting message

commit ...output omitted...
Author: ...output omitted...
Date:   Fri Sep 25 17:11:18 2020 +0200

    extracted cli argument to variable

commit ...output omitted...
Author: ...output omitted...
Date:   Fri Sep 25 14:39:19 2020 +0200

    added parameter to greeting message

commit ...output omitted...
Author: ...output omitted...
```

```
Date:   Fri Sep 25 13:37:44 2020 +0200
```

```
added helloworld
```

If Git shows the output by using a pager, then press q to quit the log screen.

- ▶ **8.** Return to the workspace directory:

```
[user@host hello-remote]$ cd ..  
[user@host DO400]$
```

This concludes the guided exercise.

Releasing Code

Objectives

After completing this section, you should be able to version and publish code with Git.

Constructing Releases

As a software developer, one of your main goals is to deliver functional versions of your application to your users. Those functional versions are called *releases*.

The releasing process involves multiple steps, such as planning the release, running quality checks, and generating documentation. This section focuses on creating and publishing a release with Git.

Naming Releases

One of the most important steps involved in the release of a new version of your application is appropriately versioning and naming the release. Naming a release is assigning a unique name to a specific point in the commit history of your project. There is no industry standard for how to uniquely name a release, but there are multiple version naming conventions you can follow. One of the most used naming conventions is *Semantic Versioning*.

Semantic Versioning

The semantic version specification is a set of rules that indicate how version numbers are assigned, and increased. This specification names releases with a series of 3 sets of numbers in the format `MAJOR.MINOR.PATCH`.

The rules defined in the specification are the following:

- The `MAJOR` version increases when you make incompatible API changes.
- The `MINOR` version increases when you add features that are backwards compatible.
- The `PATCH` version increases when you add bug fixes that are backwards compatible.

A `MAJOR` version of zero implies the project is in early stages.

By following the semantic versioning, you are communicating the type of changes included in your releases.

Creating Releases with Git

When Git is your version control system, you can use Git tags to uniquely name your releases or versions. In Git, a *tag* is a static pointer to a specific moment in the commit history, such as releases or versions. Unlike branches, the tag pointer is not modified over time.

Git supports two different types of tags, **annotated** and **lightweight**. The main difference between the two types is the amount of metadata they store.

In lightweight tags, Git stores only the name and the pointer to the commit. The `git tag` command creates a lightweight tag.

```
[user@host ~]$ git tag 1.2.3
```

Git stores annotated tags as full objects, which include metadata, such as the date, the details of the user who created the tag, and a comment. The `git tag` command with the `-a` option creates an annotated tag.

```
[user@host ~]$ git tag -a 1.2.3
```

After you have your code with a tag assigned, you can distribute it to your users. The most popular Git forges include features to help you with the creation and distribution of your releases. The process of creating a release in GitHub is very simple. You must specify a tag, write a name for the release, write a description, and append any artifact you want to include in the release.

Defining Development Workflows

Developers love finding reusable solutions to problems, and the development process is a recurring problem in their day-to-day life. For that reason, you can follow one of the many workflows available to guide your development process. A workflow consists of a defined set of rules and processes, which guide your development process.

Most of the popular development workflows share the same core idea. They want to facilitate the collaboration between the developers involved in a project. In some cases, this collaboration includes the use of branches, pull requests, and *forks*.



Note

A fork is an independent copy of a repository, including branches. You can use forks to contribute to projects. Another use of forks is to start an independent line of development. In this case, you use the copied repository as the base of your changes, and continue development independently.

Trunk-based Development

In this workflow, developers avoid the creation of long-lived branches. Instead, they work directly on a shared branch named `main`, or use short-lived branches to integrate the changes. When developers finish a feature or a fix, they merge all changes to the `main` branch. This workflow requires good test coverage to reduce the risk of introducing breaking changes, and more upfront developer communication.

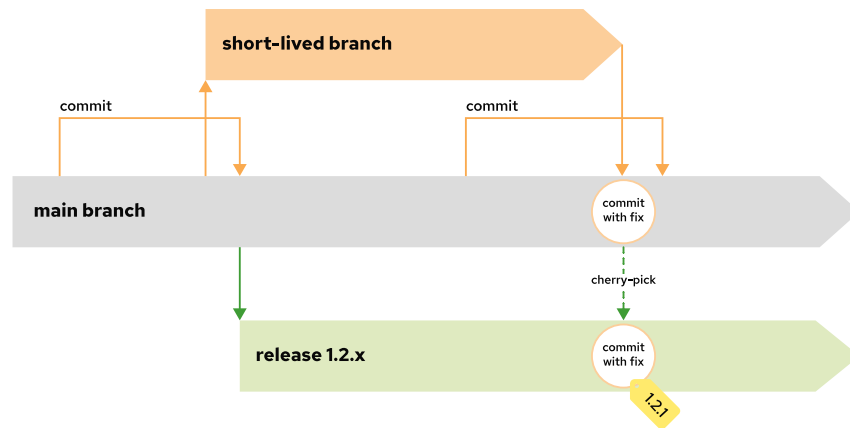


Figure 2.23: Branching structure in Trunk-based Development with a release branch

Depending on the release cadence of your project, you can perform releases from the `main` branch or from a release branch. In this workflow, any kind of development in release branches is forbidden. To update release branches, you cherry-pick the commits you want to incorporate to them, and tag the new release version.



Note

Cherry-picking is choosing a commit by its hash and appending it to the current working HEAD. The Git subcommand `git cherry-pick` allows you to pick a commit from a branch and apply it to another.

This flow works well with CI/CD practices because all the finished work is in the `main` branch. For that reason, trunk-based development is one of the preferred workflows in teams that embrace DevOps principles.

Feature Branching

In this workflow, developers work in dedicated branches, and the `main` branch only contains the stable code. In contrast to Trunk-based Development, the Feature Branching workflow does not advise against the use of long-lived branches.

Developers create a new long-lived branch from the `main` branch every time they start working on a new feature. The developer merges the feature branch into the `main` as soon as they finish the work. This encapsulation into branches facilitates the collaboration with other developers.

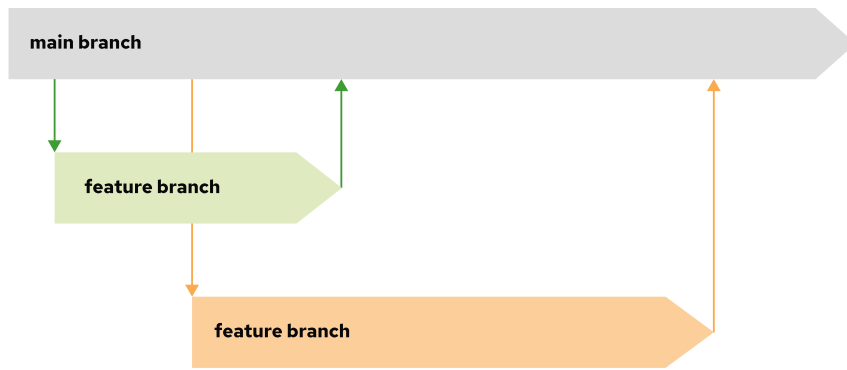


Figure 2.24: Branching structure in Feature Branching

Feature Branching is a simple set of rules that other workflows use to manage their branching models.

Git Flow

Git Flow is a well-known development workflow created in 2010 and inspired by the Feature Branching workflow.

Usually, this workflow has a central repository, which acts as the single source of truth. The central repository holds the following principal branches:

- `main`: this branch contains all the production-ready code.
- `develop`: this branch contains all the changes for the next release.

This workflow uses supporting branches with specific purposes for features, releases and fixes. A feature branch uses the `develop` branch as the starting point. When the work is done in a feature branch, you merge back this branch with the `develop` branch.

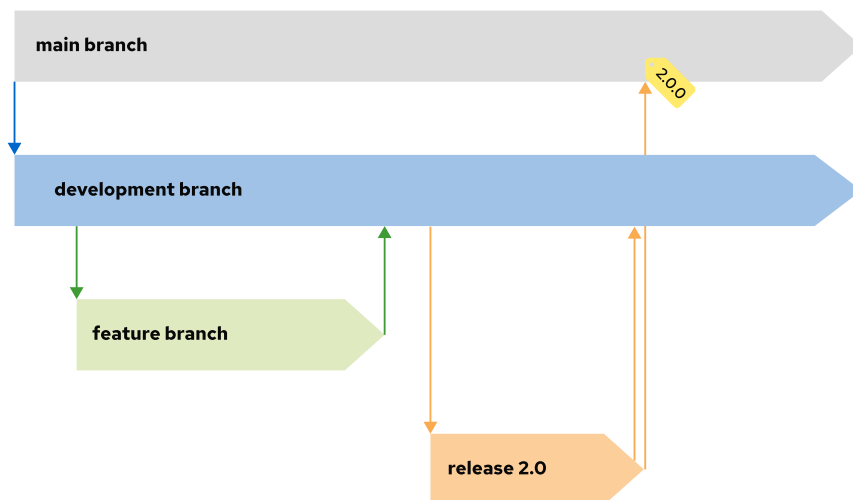


Figure 2.25: Branching structure in Git Flow

Working with release branches differs from working with feature branches. A release branch uses the `develop` branch as the starting point. You can only add minor bug fixes, and small meta-

data changes to this branch. When you finish all the work in a release branch, you must merge the branch in the `development` and `main` branches. After the merge, you must add a tag with a release number.

GitHub Flow

The GitHub Flow is a lightweight and flexible variation of the Feature Branching workflow. It tries to make the process of releasing code as simple as possible.

This workflow has a main branch named `main`, and is the single source of truth. The `main` branch contains code in a deployable state and all the development happens in separated branches.

The main difference with the Feature Branching workflow is the use of pull requests. Developers must open pull requests when they want to merge code changes. Other developers analyze the code in the pull request, and give their feedback.

You can only merge reviewed and approved code, especially if the merge is with the `main` branch. After the approval of your pull request, you can deploy the approved code to test it in the production environment. As soon as you validate the changes, you must merge the code into the main branch.

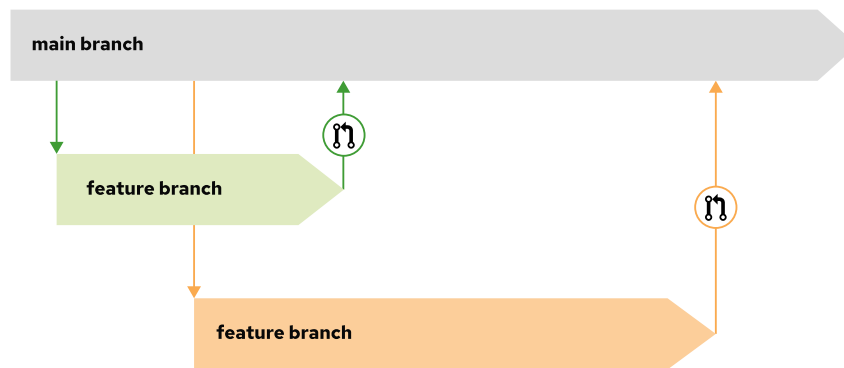


Figure 2.26: Branching structure in GitHub Flow



Note

This workflow does not give guidance about the use of specific branches for releases.

The GitHub Flow and the Trunk-based Development workflows are very similar. The main difference between them is where the release originates. In GitHub Flow, the release occurs on branches. In Trunk-based Development, it usually occurs on the `main` branch.

GitLab Flow

The GitLab Flow is a simple workflow very similar to the GitHub Flow. In this workflow, the main branch is the single source of truth, and all the code in this branch must be in a deployable state. You must merge any code changes first into the main branch.

One of the main goals of the GitLab Flow is to easily work with different deployment environments. To accomplish that, this workflow establishes that you must have a branch for each one of your deployment environments. Those branches reflect the deployed code in each

environment. You can deploy a new version of your application by merging the main branch into the environment branch.

When you make bug fixes, first you merge the fixes to the main branch, and then cherry-pick them in the different branches.

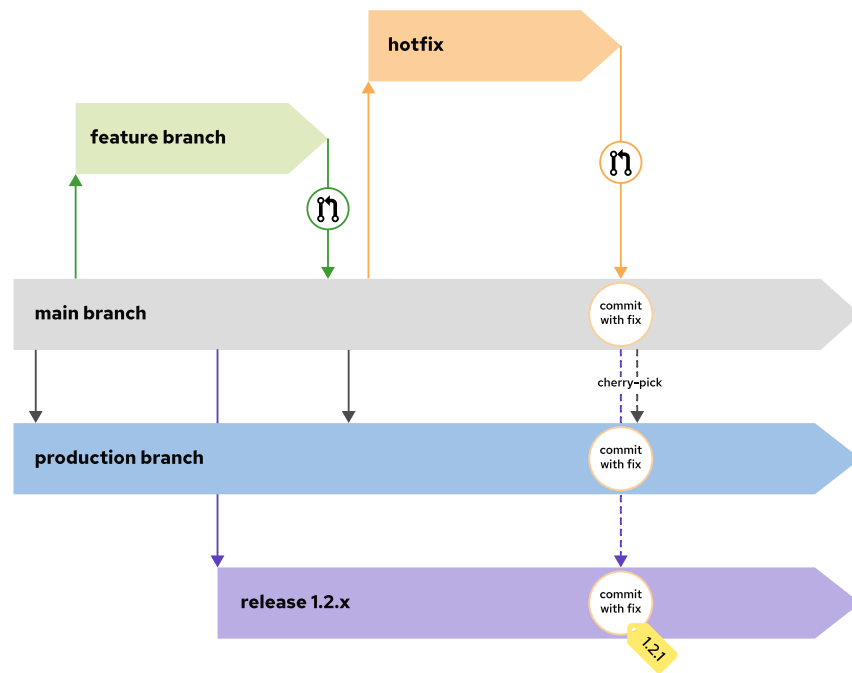


Figure 2.27: Branching structure in GitLab Flow

This workflow also enables you to work efficiently with release branches. You create a release branch by using the main branch as a starting point. After the creation of the release branch, you only add bug fixes to this branch by cherry-picking them. You must tag and increase the release version every time you add a fix to the release branch.



References

Semantic Versioning

<https://semver.org>

Trunk-based Development

<https://trunkbaseddevelopment.com>

Git Feature Branch Workflow

<https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>

A successful Git branching model

<https://nvie.com/posts/a-successful-git-branching-model>

Understanding the GitHub Flow

<https://guides.github.com/introduction/flow/>

Introduction to GitLab Flow

https://docs.gitlab.com/ee/topics/gitlab_flow.html

► Guided Exercise

Releasing Code

In this exercise you will learn how to use tags, releases, and forks.

To set up the scenario necessary to complete this exercise, first you must create a new GitHub organization with your GitHub account. Next, you will create a repository in the organization by forking the source code we provide at <https://github.com/RedHatTraining/D0400-apps-external>. The repository forked in your organization will simulate a third-party project in which to contribute.

With the organization set up as a hypothetical third-party project, you will practice a common contribution workflow, forking the third-party project into your username account and creating pull requests back to the third-party project to propose code changes.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to create tags, publish releases, and use forks to contribute to other repositories.

Before You Begin

To perform this exercise, ensure you have Git installed on your computer and access to a command line terminal.



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start every guided exercise from this folder.

Instructions

- 1. Create your own GitHub organization. This organization will host your simulated third-party project.
 - 1.1. Open a web browser and navigate to <https://github.com>.
 - 1.2. Sign in with your GitHub credentials. If you do not have an account, then create one.
 - 1.3. Click + → **New organization** in the upper right of the window.
 - 1.4. Choose the Free plan. Click **Join for free**.

- 1.5. Configure your new organization. Fill in the **Organization account name** field with the `YOUR_GITHUB_USER-do400-org` value. Type your email in the **Contact email** field. Select **My personal account** as the owner. Click **Next**.

Tell us about your organization

Set up your team

Organization account name *

your-user-do400-org ✓

This will be the name of your account on GitHub.
Your URL will be: <https://github.com/your-user-do400-org>.

Contact email *

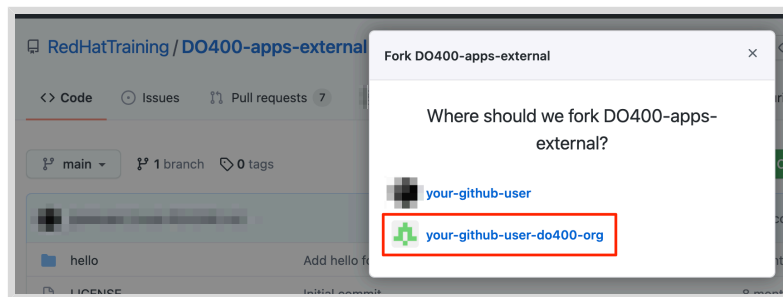
your-email@example.com ✓

This organization belongs to: *

☒ **My personal account**
I.e., your-user (Your User)

☐ **A business or institution**
For example: GitHub, Inc., Example Institute, American Red Cross

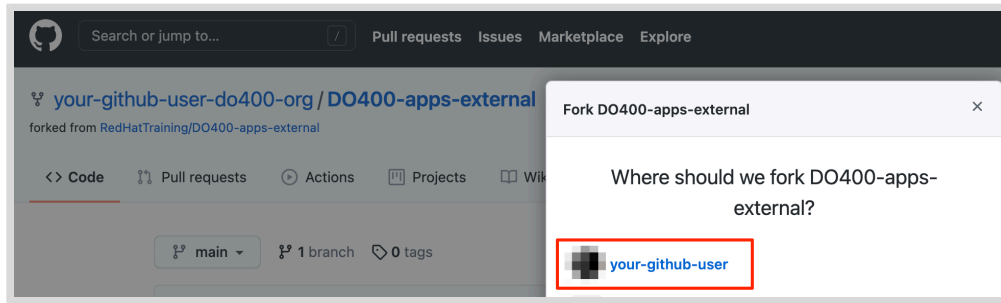
- 1.6. In the confirmation screen, click **Complete setup**. You must provide your password to complete this step.
- ▶ 2. Create a repository in your organization by forking the `DO400-apps-external` repository. The newly created repository in your organization simulates a third-party project in which to contribute.
 - 2.1. In a web browser, navigate to the repository located at <https://github.com/RedHatTraining/DO400-apps-external>.
 - 2.2. Click **Fork** in the upper right and select the `YOUR_GITHUB_USER-do400-org` organization as the namespace. This creates the `YOUR_GITHUB_USER-do400-org/DO400-apps-external` repository.



At this point, you have set up the scenario to contribute to a third-party project. You will treat the repository you have just created as the *upstream* repository, meaning that you will consider this repository as the third-party project you send your contributions to.

- ▶ 3. Fork the upstream repository to start contributing to the upstream project. Fork the `YOUR_GITHUB_USER-do400-org/DO400-apps-external` repository to your username account and clone the forked repository located at https://github.com/YOUR_GITHUB_USER/DO400-apps-external.

- 3.1. From the upstream repository main page located at https://github.com/YOUR_GITHUB_USER-do400-org/DO400-apps-external, click Fork and select your username as the namespace.



The new fork is located at https://github.com/YOUR_GITHUB_USER/DO400-apps-external. In this exercise, you will contribute to the upstream by sending pull requests from this fork to the organization upstream repository.

- 3.2. Click the **Code** button and copy the HTTPS URL.
- 3.3. Switch to the command line and navigate to your workspace folder.

```
[user@host ~]$ cd DO400
```

- 3.4. Clone the repository by using the `git clone` command and the copied URL. Enter your GitHub username and personal access token when prompted.

```
[user@host DO400]$ git clone \
https://github.com/YOUR_GITHUB_USER/DO400-apps-external
Username for 'https://github.com': YOUR_GITHUB_USER
Password for 'https://
your_github_user@github.com': YOUR_GITHUB_PERSONAL_ACCESS_TOKEN
Cloning into 'DO400-apps-external'...
remote: Enumerating objects: 151, done.
remote: Total 151 (delta 0), reused 0 (delta 0), pack-reused 151
Receiving objects: 100% (151/151), 149.14 KiB | 208.00 KiB/s, done.
Resolving deltas: 100% (58/58), done.
```



Important

Make sure that you clone the repository from your username account and not from your organization.

By default, the `git clone` creates a folder with the same name as the repository and clones the code inside the created folder. You can change the folder by adding the folder name as a parameter of the `git clone` command.

- ▶ 4. Create a tag. Next, create a release by using the created tag.
 - 4.1. Navigate to the repository folder and create the `1.0.0` tag by using the `git tag` command:


```
[user@host D0400]$ cd D0400-apps-external
[user@host D0400-apps-external]$ git tag 1.0.0
```

4.2. Use the `git tag` command to verify that the tag has been created:

```
[user@host D0400-apps-external]$ git tag
1.0.0
```



Note

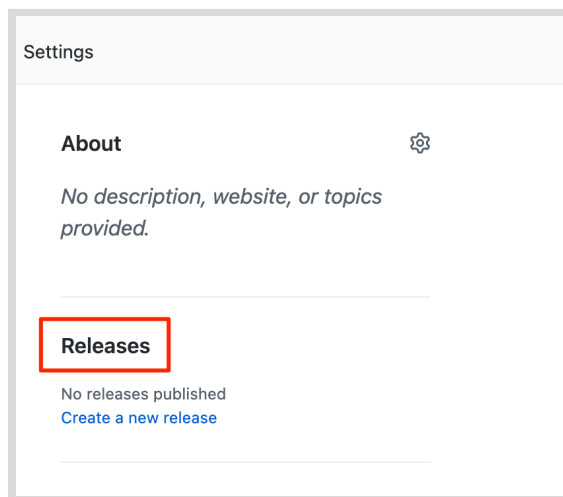
If Git shows the output by using a pager, press `q` to quit the pager.

4.3. Use `git push origin --tags` to push the tag to your username fork on GitHub.

By default, Git does not push tags to the remote. To push tags, you must use the `--tags` parameter. If prompted, enter your GitHub username and personal access token.

```
[user@host D0400-apps-external]$ git push origin --tags
Username for 'https://github.com': YOUR_GITHUB_USER
Password for 'https://
your_github_user@github.com': YOUR_GITHUB_PERSONAL_ACCESS_TOKEN
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/your_github_user/D0400-apps-external.git
 * [new tag]          1.0.0 -> 1.0.0
```

4.4. Switch back to the web browser and navigate to the main GitHub page for your fork located at `https://github.com/YOUR_GITHUB_USER/D0400-apps-external`. Click the **Releases** link in the right pane.



4.5. On the releases page, click **Draft a new release**.

**Note**

Normally you do not create releases in a forked repository.

You should use forks as a tool to contribute to the upstream repository. Upstream owners can later create releases with your contributions.

- 4.6. Start typing `1.0.0` in the **Tag version** field and select `1.0.0`. This is the tag that you just created. Do not change the **Target: main** selector.
- 4.7. Enter a value in the **Release title** field. The **Existing tag** label should appear under the tag field, confirming that you are using the `1.0.0` tag that you created.

Finally, click **Publish release** to create the release. This takes you to the newly created release page.

- 5. Create and merge a pull request from your username fork (origin) to your organization repository (upstream).

- 5.1. In your terminal, create a new branch named `hello-redhat`:

```
[user@host D0400-apps-external]$ git checkout -b hello-redhat
Switched to a new branch 'hello-redhat'
```

- 5.2. Inside the `hello` folder, create a new file `hellorh.py` with the following content:

```
print("Hello Red Hat!")
```

- 5.3. Stage and commit the `hello/hellorh.py` file:

```
[user@host D0400-apps-external]$ git add hello
[user@host D0400-apps-external]$ git commit -m "added hello Red Hat"
[hello-redhat 60db2e8] added hello Red Hat
1 file changed, 1 insertion(+)
create mode 100644 hello/hellorh.py
```

- 5.4. Push the new branch to the username fork by using `git push -u origin hello-redhat`. The `-u` parameter is an alias of `--set-upstream` and it is used to set the upstream branch. Enter your GitHub username and password if prompted.

```
[user@host D0400-apps-external]$ git push -u origin hello-redhat
...output omitted...
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 308 bytes | 308.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'hello-redhat' on GitHub by visiting:
remote:   https://github.com/your_github_user/D0400-apps-external/pull/new/
hello-redhat
remote:
To https://github.com/your_github_user/D0400-apps-external.git
 * [new branch]      hello-redhat -> hello-redhat
Branch 'hello-redhat' set up to track remote branch 'hello-redhat' from 'origin'.
```

- 5.5. Switch back to the web browser and navigate to the GitHub page of your fork located at https://github.com/YOUR_GITHUB_USER/D0400-apps-external. Click **Pull requests** to navigate to the pull requests page. In the pull requests page, click **New pull request**.
- 5.6. In the branches selection area, click the **compare: main** selector to select the `hello-redhat` branch. Observe how you are merging your changes from the `hello-redhat` branch of your username fork (`YOUR_GITHUB_USER/D0400-apps-external`) into the `main` branch of the organization upstream repository (`YOUR_GITHUB_USER-do400-org/D0400-apps-external`). Next, click the **Create pull request** button to open the pull request creation form.

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base repository: your-user-do400-org/D0400-apps-external

base: main

←

head repository: your-user/D0400-apps-external

compare: hello-redhat



Important

Make sure you do not select RedHatTraining/D0400-apps-external as the base repository.

- 5.7. In the pull request creation form, click **Create pull request** to submit the form and open the pull request.



Note

When creating a pull request, try to set meaningful titles and descriptions to add better context to your changes.

- 5.8. Review your changes and merge the pull request into the `main` branch of the upstream organization repository. Click **Merge pull request**. Click **Confirm merge** on the confirmation form that shows up.

**Note**

You should not normally merge your own pull requests.

In most cases, pull requests are merged by repository owners or maintainers. Consider that, when merging this pull request, you have acted as the owner of the upstream repository.

- 5.9. A **Delete branch** button is displayed immediately after the merge. The branch `hello-redhat` is unnecessary now because all of its changes have been merged into the `main` branch. Click **Delete branch** to delete the `hello-redhat` remote branch.

► **6.** Pull changes from your username fork. Check that the `main` branch is not updated.

- 6.1. On the command line, switch to the `main` branch:

```
[user@host D0400-apps-external]$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

- 6.2. Pull the changes from the `origin` remote. The `origin` remote refers to the repository forked in your username account (`YOUR_GITHUB_USER/D0400-apps-external`). Enter your GitHub username and password if prompted.

```
[user@host D0400-apps-external]$ git pull origin main
...output omitted...
From https://github.com/your_github_user/D0400-apps-external
* branch          main          -> FETCH_HEAD
Already up to date.
```

- 6.3. Run `git log` and notice that your changes from the `hello-redhat` branch were not incorporated.

```
[user@host D0400-apps-external]$ git log
commit c34ac6fcb5eea6ed4c62b665f0bb3b6c18f9a579 (HEAD -> main, tag: 1.0.0, origin/main, origin/HEAD)
...output omitted...
```

Note how the last commit in the `main` branch is still the commit tagged as `1.0.0`. This is because you merged the pull request to the **upstream** repository (`YOUR_GITHUB_USER-do400-org/D0400-apps-external`) and not the username fork (`YOUR_GITHUB_USER/D0400-apps-external`).

► **7.** Configure and pull from the `upstream` remote.

- 7.1. In the web browser, navigate to the main page of your organization upstream repository, located at: https://github.com/YOUR_GITHUB_USER-do400-org/D0400-apps-external. Click the **Code** button and copy the HTTPS URL.

- 7.2. Configure a new remote called `upstream` to point to your organization repository:

```
[user@host D0400-apps-external]$ git remote add upstream \
https://github.com/YOUR_GITHUB_USER-do400-org/D0400-apps-external.git
```

- 7.3. Pull from the `upstream` remote by using `git pull upstream main -p`. Enter your GitHub username and password if prompted.

```
[user@host D0400-apps-external]$ git pull upstream main
...output omitted...
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From https://github.com/your_github_user-do400-org/D0400-apps-external
 * branch          main      -> FETCH_HEAD
 * [new branch]     main      -> upstream/main
Updating 6459e50..d4492a5
Fast-forward
 hello/hellorh.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 hello/hellorh.py
```

- 7.4. Run `git log` to see the local `main` branch has been updated:

```
[user@host D0400-apps-external]$ git log
commit d4492a5546f3088af378ef287dd272970ac3eb5d (HEAD -> main, upstream/main)
Merge: 6459e50 60db2e8
Author: Your User Name <your.email@example.com>
Date: Thu Oct 1 12:54:59 2020 +0200

    Merge pull request #1 from your_github_user/hello-redhat

    added hello Red Hat

commit 60db2e8b62bad2320e66aa94e147c66aefc71139 (origin/hello-redhat, hello-redhat)
Author: Your User Name <your.email@example.com>
Date: Thu Oct 1 12:38:01 2020 +0200

    added hello Red Hat

...output omitted...
```

Notice how both the local and the upstream `main` branches point to the same commit.

► 8. Create a new release from GitHub.

- 8.1. Push the changes on `main` to the forked repository with `git push origin main`. Enter your GitHub username and password if prompted.

```
[user@host D0400-apps-external]$ git push origin main
...output omitted...
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/your_github_user/D0400-apps-external.git
6459e50..d4492a5  main -> main
```

- 8.2. Refresh the GitHub page for your username fork located at https://github.com/YOUR_GITHUB_USER/D0400-apps-external. You should see the latest commit and changes.
- 8.3. Click **Releases** on the right.
- 8.4. On the releases page, click **Draft a new release**.

**Note**

Normally you should not create releases in a forked repository.

- 8.5. Enter **2.0.0** in the **Tag version** field and enter a value in the **Release title** field.
- 8.6. Click **Publish release** to create the release. You are taken to your release's page.
- 8.7. On the command line, run `git pull --tags`. Enter your GitHub username and password if prompted.

```
[user@host D0400-apps-external]$ git pull --tags
...output omitted...
From https://github.com/your_github_user/D0400-apps-external
* [new tag]          2.0.0      -> 2.0.0
Already up to date.
```

- ▶ 9. Run `git log` to see your updated changes on `main` and the new **2.0.0** tag:

```
[user@host D0400-apps-external]$ git log
commit d4492a5546f3088af378ef287dd272970ac3eb5d (HEAD -> main, tag: 2.0.0,
    upstream/main, origin/main, origin/HEAD)
Merge: 6459e50 60db2e8
Author: Your User Name <your.email@example.com>
Date: Thu Oct 1 12:54:59 2020 +0200

    Merge pull request #1 from your_github_user/hello-redhat

    added hello Red Hat

...output omitted...
```

- ▶ 10. Clean up your local files and the remote repositories and organization.

- 10.1. Remove your local `D0400-apps-external` folder

```
[user@host D0400-apps-external]$ cd ..
[user@host D0400]$ rm -rf D0400-apps-external
```

**Important**

Windows users must run the preceding command in PowerShell as follows:

```
PS C:\Users\user\D0400> rm -Recur -Force D0400-apps-external
```

10.2. Remove the downstream and upstream repositories.

Go to your GitHub repository settings page at https://github.com/YOUR_GITHUB_USER/D0400-apps-external/settings.

Scroll to the **Danger Zone** at the bottom and click **Delete this repository**.

Type the repository name in the popup message and confirm the deletion clicking **I understand the consequences, delete this repository**. If requested, type your password to confirm your identity.

Repeat the same steps for the upstream repository at https://github.com/YOUR_GITHUB_USER-do400-org/D0400-apps-external/settings

10.3. Remove the organization.

Enter the organization settings at https://github.com/organizations/YOUR_GITHUB_USER-do400-org/settings/profile.

Scroll to the **Danger Zone** at the bottom and click **Delete this organization**.

Type the repository name in the popup message and confirm the deletion.

This concludes the guided exercise.

► Lab

Integrating Source Code with Version Control

In this lab, you will open pull requests to a new Git repository and fix a merge conflict within it.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Create a local repository
- Create a GitHub repository
- Create branches
- Commit changes
- Open pull requests in GitHub
- Fix merge conflicts

Before You Begin

To perform this exercise, ensure you have:

- Git installed
- A GitHub account
- A web browser, such as Firefox or Chrome

Make sure you start this lab from your workspace folder.

Instructions

1. Configure your name and email in Git.
You can skip this step if you already have your name and email configured in Git.
2. Create a new local Git repository named `do400-git-lab` and commit a new file named `README.md` with the following contents:

```
# do400-git-lab
```

```
This is an example project repository for the D0400 course.
```

3. Create an empty repository on GitHub named `do400-git-lab` and push the local repository there.
4. Change the `README.md` file to append a new line. Commit the changes to a new branch named `readme-update`. Push the new branch to the GitHub remote.

After you are done, the `README.md` file should contain the following:

```
# do400-git-lab

This is an example project repository for the D0400 course.
This repository is only a test.
```

5. Open a pull request from the `readme-update` branch to the `main` branch. Do not merge it yet.
6. For a second time, change the `README.md` file to append a different line. Commit the changes to another new branch named `readme-example`.

This branch should be based off of the `main` branch, *not* the `readme-update` branch from the previous steps. Push the new branch to the GitHub remote.

After you are done, the `README.md` file should contain the following:

```
# do400-git-lab

This is an example project repository for the D0400 course.
This repository is a simple example.
```

7. Open and merge a pull request from the `readme-example` branch to the `main` branch. Once merged, delete the branch.
8. Navigate to the first pull request. Check that this pull request now has conflicts with the `main` branch. Using the Git command-line interface, fix the conflicts for the first pull request.

After resolving the conflicts, the `README.md` file should contain the following:

```
# do400-git-lab

This is an example project repository for the D0400 course.
This repository is a simple test example.
```

Note that there are several specific ways to accomplish this step. The solution provided is only one such way.

9. Sync changes from both merged pull requests to your local `main` branch so that it is ready for future development.

This concludes the lab.

► Solution

Integrating Source Code with Version Control

In this lab, you will open pull requests to a new Git repository and fix a merge conflict within it.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (`) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Create a local repository
- Create a GitHub repository
- Create branches
- Commit changes
- Open pull requests in GitHub
- Fix merge conflicts

Before You Begin

To perform this exercise, ensure you have:

- Git installed
- A GitHub account
- A web browser, such as Firefox or Chrome

Make sure you start this lab from your workspace folder.

Instructions

1. Configure your name and email in Git.

You can skip this step if you already have your name and email configured in Git.

- 1.1. Open a new terminal window on your workstation, and use the `git config` command to check whether the values are set. If they are set, then skip the remainder of this step.

```
[user@host D0400]$ git config --get user.name
[user@host D0400]$ git config --get user.email
```

- 1.2. Use the `git config` command to configure your name.

```
[user@host D0400]$ git config --global user.name "YOUR USER NAME"
```

- 1.3. Use the `git config` command to configure your user email.

```
[user@host D0400]$ git config --global user.email YOUR@USER.EMAIL
```

- 1.4. Use the `git config` command to review your identity settings.

```
[user@host D0400]$ git config --get user.name
Your User Name
[user@host D0400]$ git config --get user.email
your@user.email
```

2. Create a new local Git repository named `do400-git-lab` and commit a new file named `README.md` with the following contents:

```
# do400-git-lab

This is an example project repository for the D0400 course.
```

- 2.1. Create a new directory to house the repository and navigate to it.

```
[user@host D0400]$ mkdir do400-git-lab
[user@host D0400]$ cd do400-git-lab
[user@host do400-git-lab]$
```

- 2.2. Initialize the directory as a Git repository.

```
[user@host do400-git-lab]$ git init
Initialized empty Git repository in /path/to/workspace/do400-git-lab/.git/
```

- 2.3. Create the file using your editor or from the command line. For example:

```
[user@host do400-git-lab]$ cat << EOF > README.md
> # do400-git-lab
>
> This is an example project repository for the D0400 course.
> EOF
```



Important

Windows users must also run the preceding command but in PowerShell as follows:

```
PS C:\Users\user\D0400\do400-git-lab> @"
>> # do400-git-lab
>> This is an example project repository for the D0400 course.
>> "@ | Tee-Object -FilePath "README.md"
```

- 2.4. Stage and commit the `README.md` file. Note that the hash for your commit will be different.

```
[user@host do400-git-lab]$ git add README.md
[user@host do400-git-lab]$ git commit -m 'added README'
[master (root-commit) 078df7f] added README
1 file changed, 3 insertions(+)
create mode 100644 README.md
```

2.5. Rename the branch from `master` to `main`.

```
[user@host do400-git-lab]$ git branch -m master main
```

You may optionally skip this step. If so, be sure to replace `main` with `master` in the remaining steps.



Note

For Red Hat's statement on conscious language efforts, please see <https://www.redhat.com/en/blog/update-red-hats-conscious-language-efforts>.

3. Create an empty repository on GitHub named `do400-git-lab` and push the local repository there.
 - 3.1. In a web browser, navigate to the GitHub home page at <https://github.com>. Be certain that you are logged in.
 - 3.2. Click **New** in the left pane.
 - 3.3. Make sure your GitHub user name is selected as the **Owner**, and enter `do400-git-lab` for **Repository name**. Do not change any of the other values. You may optionally set the repository as private.
 - 3.4. Click **Create repository**.
 - 3.5. Select **HTTPS** and copy the URL to your clipboard.
 - 3.6. Using the copied URL, add the GitHub repository as a remote named `origin` to your local repository.

```
[user@host do400-git-lab]$ git remote add origin https://
github.com/YOUR_GITHUB_USERNAME/do400-git-lab.git
```

- 3.7. Push the `main` branch to the remote, setting the remote as the default push location for the branch.

```
[user@host do400-git-lab]$ git push -u origin main
Username for 'https://github.com': YOUR_GITHUB_USERNAME
Password for 'https://your_github_username@github.com':
...output omitted...
To https://github.com/your_github_username/do400-git-lab.git
* [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

If prompted for credentials, then provide your username and personal access token.

4. Change the `README.md` file to append a new line. Commit the changes to a new branch named `readme-update`. Push the new branch to the GitHub remote.

After you are done, the `README.md` file should contain the following:

```
# do400-git-lab

This is an example project repository for the D0400 course.
This repository is only a test.
```

- 4.1. Update the file using your editor or from the command line. For example:

```
[user@host do400-git-lab]$ echo "This repository is only a test." >> README.md
```

- 4.2. Create and switch to a new branch named `readme-update`.

```
[user@host do400-git-lab]$ git checkout -b readme-update
Switched to a new branch 'readme-update'
```

- 4.3. Stage and commit the changes.

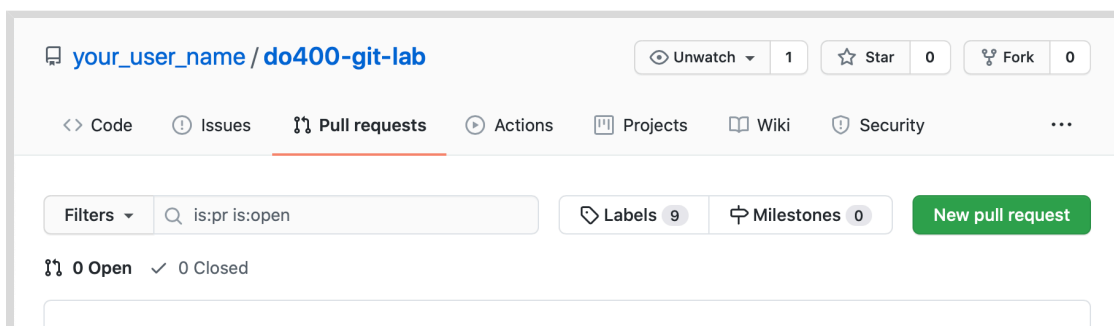
```
[user@host do400-git-lab]$ git add README.md
[user@host do400-git-lab]$ git commit -m 'added test repository notice'
[readme-update d4009df] added test repository notice
1 file changed, 1 insertion(+)
```

- 4.4. Push the new branch to GitHub, setting the remote as the default push location for the branch.

```
[user@host do400-git-lab]$ git push -u origin readme-update
...output omitted...
To https://github.com/your_github_username/do400-git-lab.git
 * [new branch]      readme-update -> readme-update
Branch 'readme-update' set up to track remote branch 'readme-update' from
'origin'.
```

5. Open a pull request from the `readme-update` branch to the `main` branch. Do not merge it yet.

- 5.1. From the **Pull Requests** tab of the GitHub repository, click **New pull request**.



- 5.2. Leave `Base` set to `main`, but set `Compare` to `readme-update`. Your changes to `README.me` should appear.
- 5.3. Click **Create pull request**.
- 5.4. The title for the pull request should be preset to the commit message you provided in a previous step. If not, then provide a memorable title, such as the commit message.
- 5.5. Click **Create pull request**. You are taken to the details page for the pull request once it is created.

Do not merge it yet.

6. For a second time, change the `README.md` file to append a different line. Commit the changes to another new branch named `readme-example`.
This branch should be based off of the `main` branch, *not* the `readme-update` branch from the previous steps. Push the new branch to the GitHub remote.
After you are done, the `README.md` file should contain the following:

```
# do400-git-lab

This is an example project repository for the D0400 course.
This repository is a simple example.
```

- 6.1. Check out the `main` branch.

```
[user@host do400-git-lab]$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

- 6.2. Update the file using your editor or from the command line. For example:

```
[user@host do400-git-lab]$ echo "This repository is a simple example." \
>> README.md
```

- 6.3. Create and switch to a new branch named `readme-example`.

```
[user@host do400-git-lab]$ git checkout -b readme-example
Switched to a new branch 'readme-example'
```

- 6.4. Stage and commit the changes.

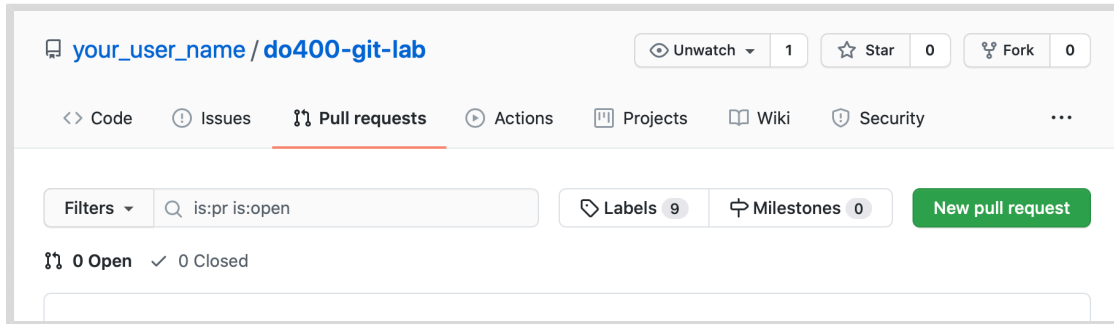
```
[user@host do400-git-lab]$ git add README.md
[user@host do400-git-lab]$ git commit -m 'added example repository notice'
[readme-example 0c67dea] added example repository notice
1 file changed, 1 insertion(+)
```

- 6.5. Push the new branch to GitHub, setting the remote as the default push location for the branch.

```
[user@host do400-git-lab]$ git push -u origin readme-example
...output omitted...
To https://github.com/your_github_username/do400-git-lab.git
 * [new branch]      readme-example -> readme-example
Branch 'readme-example' set up to track remote branch 'readme-example' from
'origin'.
```

7. Open and merge a pull request from the `readme-example` branch to the `main` branch. Once merged, delete the branch.

7.1. From the **Pull Requests** tab of the GitHub repository, click **New pull request**.



- 7.2. Leave **Base** set to `main`, but set **Compare** to `readme-example`. Your changes to `README.me` should appear.
- 7.3. Click **Create pull request**.
- 7.4. The title for the pull request should be preset to the commit message you provided in a previous step. If not, then provide a memorable title, such as the commit message.
- 7.5. Click **Create pull request**. You are taken to the details page for the pull request once it is created.
- 7.6. Merge the pull request by clicking **Merge pull request** and then **Confirm merge**.
- 7.7. Delete the unnecessary branch by clicking **Delete branch**.
8. Navigate to the first pull request. Check that this pull request now has conflicts with the `main` branch. Using the Git command-line interface, fix the conflicts for the first pull request. After resolving the conflicts, the `README.md` file should contain the following:

```
# do400-git-lab

This is an example project repository for the D0400 course.
This repository is a simple test example.
```

Note that there are several specific ways to accomplish this step. The solution provided is only one such way.

- 8.1. Navigate to the first pull request by clicking the **Pull requests** tab and clicking the title of the pull request. Notice that the GitHub interface warns you of conflicts and disables the merge button.

**Note**

GitHub provides an interface within the browser to resolve conflicts, which is useful when minimal changes are necessary.

However, you should be familiar with using the command-line interface to resolve Git conflicts.

8.2. On the command line, make sure you have the `readme-update` branch checked out.

```
[user@host do400-git-lab]$ git checkout readme-update
Switched to branch 'readme-update'
Your branch is up to date with 'origin/readme-update'.
```

8.3. Pull the merged changes from the `main` branch on the `origin` remote.

```
[user@host do400-git-lab]$ git pull origin main
...output omitted...
Unpacking objects: 100% (1/1), 640 bytes | 640.00 KiB/s, done.
From https://github.com/your_github_username/do400-git-lab
* branch                main          -> FETCH_HEAD
   078df7f..e48fef2      main          -> origin/main
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

8.4. Edit the `README.md` and correct the file to contain the following. Be sure to remove any conflict markers, which include `<<<<<<`, `>>>>>>`, and `=====`.

```
...output omitted...
This is an example project repository for the D0400 course.
This repository is a simple test example.
```

8.5. Save, stage, and commit the `README.md` file.

```
[user@host do400-git-lab]$ git add README.md
[user@host do400-git-lab]$ git commit --no-edit
[readme-update b6043be] Merge branch 'main' ...output omitted... into readme-update
```

The `--no-edit` option instructs Git to not prompt for a commit message. Because there were merge conflicts, Git generates a useful commit message. Normally, you should provide a message.

8.6. Push the new merge commit to the GitHub remote.

```
[user@host do400-git-lab]$ git push origin readme-update
...output omitted...
To https://github.com/your_github_username/do400-git-lab.git
d4009df..b6043be  readme-update -> readme-update
```


If you used the `-u` option in previous push commands, you can instead use `git push` without any additional arguments.

- 8.7. In the browser, refresh the page for your first pull request. The conflict warning should disappear and the **Merge pull request** button should be enabled.
- 8.8. Click **Merge pull request** and confirm by clicking **Confirm merge**.
- 8.9. Delete the unnecessary branch by clicking **Delete branch**.
9. Sync changes from both merged pull requests to your local `main` branch so that it is ready for future development.
 - 9.1. On the command line, check out the `main` branch.

```
[user@host do400-git-lab]$ git checkout main
Switched to branch 'main'
```

- 9.2. Pull the `main` branch from the `origin` remote.

```
[user@host do400-git-lab]$ git pull origin main
...output omitted...
From https://github.com/your_github_username/do400-git-lab
 * branch                main          -> FETCH_HEAD
    e48fef2..8b12379      main          -> origin/main
Updating 078df7f..8b12379
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

- 9.3. Optionally, remove the unnecessary branches from your local repository, and return to the workspace directory.

```
[user@host do400-git-lab]$ git branch -d readme-update
Deleted branch readme-update (was b6043be).
[user@host do400-git-lab]$ git branch -d readme-example
Deleted branch readme-example (was 0c67dea).
[user@host do400-git-lab]$ cd ..
[user@host DO400]$
```

This concludes the lab.

Summary

In this chapter, you learned:

- Version Control Systems (VCS) record changes to files over time, and Git is the most popular VCS.
- A commit is a snapshot of your project in a specific point in time.
- A branch is a diversion from the main line of development.
- A remote is a version of your repository hosted in an external location.
- A tag is a static pointer to a specific moment in the commit history.
- You can collaborate with a team on contributions via pull requests.

Chapter 3

Implementing Unit, Integration, and Functional Testing for Applications

Goal

Describe and implement the foundational principles behind comprehensive application testing and implement unit, integration, and functional testing.

Objectives

- Describe the basics of testing and define the various types of tests.
- Implement unit tests and mock services to test applications.
- Implement back-end integration tests.
- Create tests to validate that an application meets its functional requirements.

Sections

- Describing the Testing Pyramid (and Quiz)
- Creating Unit Tests and Mock Services (and Guided Exercise)
- Creating Integration Tests (and Guided Exercise)
- Building Functional Tests (and Guided Exercise)

Lab

Implementing Unit, Integration, and Functional Testing for Applications

Describing the Testing Pyramid

Objectives

After completing this section, you should be able to describe the basics of testing and define the various types of tests.

Introducing Testing

One of the goals of DevOps is to limit opportunities for human error. Ultimately, however, Software development is subject to human errors. Mistakes in the software life cycle can lead to **defects** and **failures** in the affected software system. In the worst cases, this can result in serious economic and human damage. For example, imagine that you develop a healthcare software system, which calculates medication doses prescribed to different patients, based on the clinical history of each patient. Any error in this system could potentially lead to serious effects on the health status of the patients.



Note

There are multiple terms involved in the definition of a software defect. In theory, humans make *mistakes* or *errors*, causing *defects* or *bugs* in the software, which can potentially lead to system *failures*.

In practice, just using the *error* or *bug* terms is enough to refer to software defects.

Quality Assurance (QA) is necessary to avoid these situations, minimize the number of errors, and improve the quality of a product. In software development, the key QA technique is *Software testing*, which executes a series of validations and analyzes software applications to find errors.

Categorizing Software Testing

There are multiple approaches and techniques to software testing, but their definitions are often subject to debate, and sometimes overlap. Although knowing which one to use is not always obvious, there are generally agreed categorizations. Depending on what is tested, the scope, and the goal of the tests, some of the most well-known categorizations are the following:

Requirements Testing

Functional testing

Functional testing comprises the different disciplines and test types focused on validating the **functional requirements** of an application. For example, in an e-commerce application, functional testing could validate that users can purchase products successfully. Examples of functional testing are *unit tests*, *integration tests*, and *functional tests*. These test types will be covered in detail in this chapter.



Note

Be careful not to confuse *functional testing*, which is a software testing category, with *functional tests*, a specific type of tests.

Non-functional testing

Non-functional testing focuses on testing the system against **non-functional requirements**. For example, in web applications, a frequent non-functional requirement is the response time. Therefore, non-functional tests make sure that a web application is fast enough to meet the response time requirement. Examples of non-functional tests are **load tests**, stress tests, **security tests**, and usability tests.



Note

You can also think of functional tests as a way to test the *WHAT*, and non-functional tests as a way to test the *HOW*.

Code Execution

Static testing

Static testing does not run the application in order to test it. Instead, it performs an analysis of the source code and possibly other project files. Static testing can be either **manual** or **automatic**. **Code reviews** and **automatic code analysis** are examples of static testing.

Dynamic testing

In contrast, dynamic testing is carried out by executing the code of the application under test. The majority of testing techniques covered in this course are dynamic.

Application Internals Visibility

There are two main categories of test: *black-box tests* and *white-box tests*. Which category a particular test falls under depends on how you treat the code inside the *subject under test*. In practice, these distinctions are not always clear.

Black-box testing

With black-box testing, you test the application output without considering the application internals. Metaphorically, imagine a black or opaque box around the contents of the code, blocking visibility. Black-box testing focuses on software boundaries, and how users and external systems interact with an application.

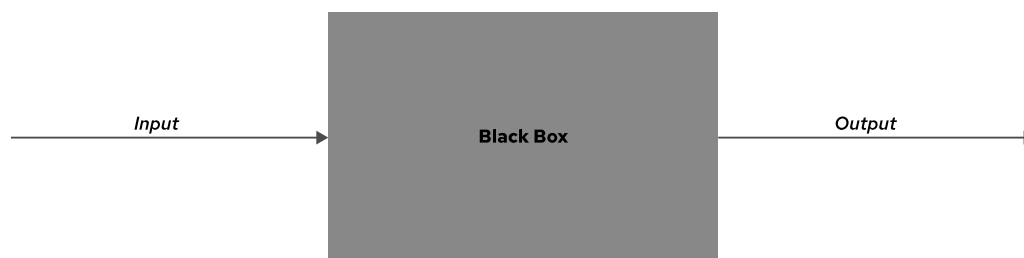


Figure 3.1: Black-box testing

White-box testing

With white-box testing, you test the inner details of an application. Again metaphorically, imagine the box around the contents is now white or translucent. White-box testing aims to validate the application from an internal point of view, considering the software details and structure, and focusing on specific inner components.

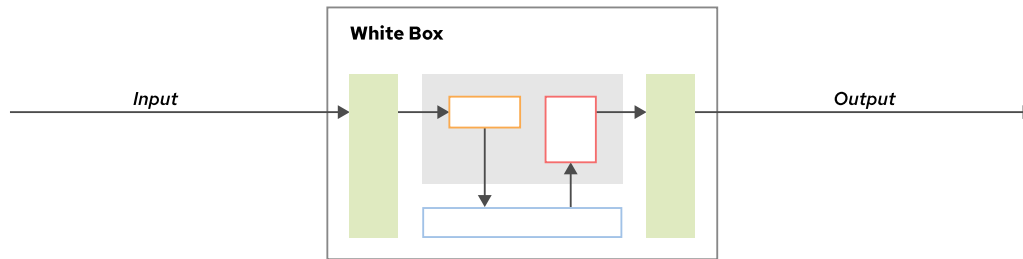


Figure 3.2: White-box testing

Automating Tests

We, as developers, have a tendency to test our work manually. First, you write a piece of code. Next, you execute the code and manually validate that the code works. If you are just developing a simple utility script, conducting some kind of research, or doing exploratory QA testing, then manual simple testing is a fair solution. As your code grows, however, it is difficult, if not impossible, to ensure the quality of an application by using manual checks only.

Developers should instead focus on automated testing as soon as possible. One benefit of automated testing is that **automated tests save time**. Developing a test might initially take more time than running the validations manually, however **writing automated tests is an investment**, which pays time-saving dividends in subsequent test runs.

Another benefit of automated testing is the repeatability in the quality checks. **Manual tests are subject to human mistakes**. Unawareness, laziness, and lack of time are factors that might disrupt the correct execution of a manual test. These factors vanish with automated tests, especially when a CI server runs them continuously.

Designing a Test

Tests are program specifications written as executable software. Developers define tests by writing the source code necessary to run the test. You can write most of the tests, regardless of the test type, by using a basic structure made of three blocks.

Basic structure of a test

1. Set up the test scenario
2. Execute the action to test
3. Validate the results

The following example shows a basic test. Despite its simplicity, note how the test includes the three mentioned blocks. This structure is commonly called the **Given-When-Then** test structure.

```
public void test_greet_says_hello() {
    String name = "Dave"; 1
    String result = greet(name); 2
    assertEquals("Hello Dave!", result); 3
}
```

- 1 Given** a specific scenario: Dave as a name.
- 2 When** you call the greet function.

③ **Then** assert that the result is `Hello Dave!`.

Describing the Testing Pyramid

To ensure the quality of your application, you need a testing strategy that combines multiple types of tests, covering the different aspects of a software system. For example, you might want to check that your business logic works as expected, validate that the integration with the storage system is correct, and check that users can correctly interact with the application by using a browser.

The testing pyramid is an approximate representation of how teams should distribute their tests.

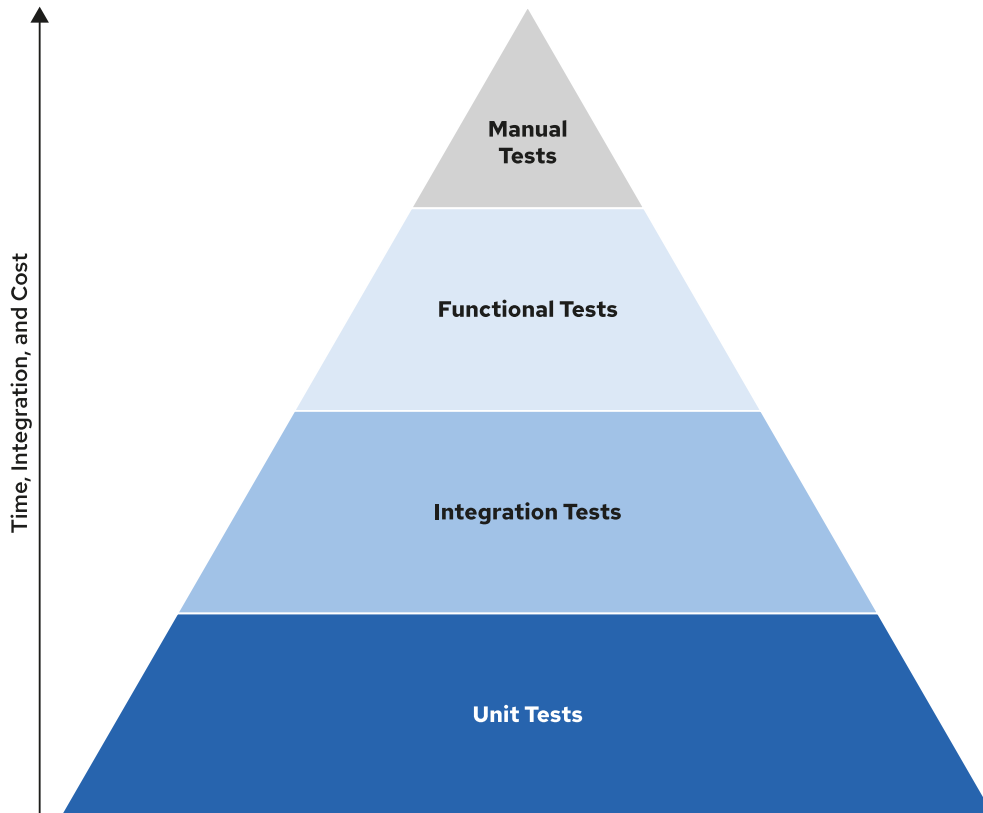


Figure 3.3: The testing pyramid

At the bottom of the pyramid, tests are faster, specific, simpler, and easier to write. As you go up the pyramid, tests cover broader scopes but become more difficult to write and maintain. Typically, tests at the highest levels return high value in terms of acceptance criteria and quality assurance, but at a high cost.

Unit Tests

The bottom of the pyramid is made of unit tests. These tests cover the application at the source code level, running very specific validations on small code *units in isolation*. Unit tests are **white-box** tests intended to be written by developers for developers, to test the core of the business rules and fully support the development cycle, usually with techniques such as Test-driven development (TDD).

**Note**

A *unit* is normally the smallest testable piece of code. Teams should agree on what a unit means for them. Some consider a unit as a function, or a class. Others consider units as modules or groups of logically related source code.

As the pyramid illustrates, unit tests must be the foundation of your testing strategy, covering most of your core logic. In fact, teams commonly use the *Code coverage* metric to measure how well tests cover their core logic.

Integration Tests

Unit tests make sure that specific code units work in isolation. However, you also must test that those units can integrate and communicate.

Integration tests check that **two or more system components work together**. Unit tests use a technique called *test doubles* to replace the dependencies of a unit with fake or simplified versions. Integration tests, in contrast, use the real dependency to test that the components integrate well together. For example, if a class depends on a database system, then a unit test will normally replace the real database with a *test double*. If, instead of using a *test double*, you use the real database, then you are writing an integration test.

For this reason, the line that separates unit from integration tests is blurry. Integration tests are more difficult to write than unit tests because normally you must set up a scenario with several components, even third-party systems.

Functional Tests

This level includes tests that evaluate the application as a whole from the perspective of the user. The terms *functional tests*, *acceptance tests*, *system tests*, and *end-to-end tests* all make reference to the act of validating that an application meets the functional requirements. Functional tests evaluate and treat the application as a **black box**. Writing and maintaining these tests has a high cost. You should carefully choose the tests that you write at this level.

Manual Tests

For a complete testing plan, you should include some manual checks. Manual tests are performed by humans, usually as a way to explore and research the behavior of the application under a pair of human eyes. These tests have the highest cost and therefore you should have less of them in your testing pyramid.

Explaining the Role of Testing in DevOps

Automated tests are closely related to continuous integration. Automation servers, such as Jenkins, are in charge of executing integration pipelines. These pipelines include testing stages to continuously test that new changes do not cause regressions. This is called *Continuous testing*.

**Note**

A regression is an error caused by a change in the system, such as the deployment of a new version. In fact, *regression testing* is a type of testing that typically runs after a system update.

The testing stages in continuous integration pipelines execute unit, integration and functional tests to check the quality of the code. Teams that introduce the pyramid of testing in their CI pipelines maximize the automation of their tests. This is the key to achieve the DevOps goals: a frequent, responsive, and reliable software delivery process, with better quality and shorter feedback loops.

Running automated tests in continuous integration pipelines also helps teams to maintain their tests. When teams make an initial effort to write automated tests but forget to continuously run them, tests break down. If tests remain broken for enough time, then the team will eventually abandon them, and all the previous testing efforts would have been a waste of time.



References

Software testing

https://en.wikipedia.org/wiki/Software_testing

The Practical Test Pyramid

<https://martinfowler.com/articles/practical-test-pyramid.html>

Glenford J. Myers, et al. The art of software testing. 3rd Edition. 2011.

► Quiz

Describing the Testing Pyramid

Choose the correct answers to the following questions:

- 1. **Which of the following is a direct benefit of automated testing?**
 - a. The source code of automatically-tested applications is cleaner.
 - b. Automated testing is an investment that saves time in the long term.
 - c. Automated testing protects applications against all potential errors.
 - d. Automated testing removes the need for code reviews.

- 2. **Which of the following two statements about the testing pyramid is correct? (Choose two.)**
 - a. Unit tests support the development process and should cover most of the business logic.
 - b. Most of the tests in a project should be functional tests.
 - c. Integration tests are generally more difficult to set up and maintain than unit tests.
 - d. Functional tests are a type of white-box testing.
 - e. There should be as many manual tests as automatic tests, because human supervision is important.
 - f. If you test a feature with a functional test, then you do not need to use unit tests in the underlying code.

- 3. **You are testing a particular class in your source code. This class depends on an external system for data storage. Assuming that you want to test that the class successfully communicates with the storage system to read and save data, which test type should you use?**
 - a. A unit test
 - b. An integration test
 - c. A functional test
 - d. A manual test
 - e. An acceptance test

- 4. **Which of the following is the best strategy to run automated tests?**
 - a. Make a team agreement to run the tests several times a day.
 - b. Make developers responsible for running the tests, when they need to.
 - c. Run tests only before and after the team releases a new version.
 - d. Set up a CI server to continuously run the tests.

- 5. You want to validate the use cases of your e-commerce web application from the user perspective. Concretely, you would like to test that the website allows users to add products to the shopping cart. Which test type should you use?
- a. A unit test
 - b. A white-box test
 - c. An integration test
 - d. A functional test

► Solution

Describing the Testing Pyramid

Choose the correct answers to the following questions:

- 1. **Which of the following is a direct benefit of automated testing?**
 - a. The source code of automatically-tested applications is cleaner.
 - b. Automated testing is an investment that saves time in the long term.
 - c. Automated testing protects applications against all potential errors.
 - d. Automated testing removes the need for code reviews.

- 2. **Which of the following two statements about the testing pyramid is correct? (Choose two.)**
 - a. Unit tests support the development process and should cover most of the business logic.
 - b. Most of the tests in a project should be functional tests.
 - c. Integration tests are generally more difficult to set up and maintain than unit tests.
 - d. Functional tests are a type of white-box testing.
 - e. There should be as many manual tests as automatic tests, because human supervision is important.
 - f. If you test a feature with a functional test, then you do not need to use unit tests in the underlying code.

- 3. **You are testing a particular class in your source code. This class depends on an external system for data storage. Assuming that you want to test that the class successfully communicates with the storage system to read and save data, which test type should you use?**
 - a. A unit test
 - b. An integration test
 - c. A functional test
 - d. A manual test
 - e. An acceptance test

- 4. **Which of the following is the best strategy to run automated tests?**
 - a. Make a team agreement to run the tests several times a day.
 - b. Make developers responsible for running the tests, when they need to.
 - c. Run tests only before and after the team releases a new version.
 - d. Set up a CI server to continuously run the tests.

- 5. You want to validate the use cases of your e-commerce web application from the user perspective. Concretely, you would like to test that the website allows users to add products to the shopping cart. Which test type should you use?
- a. A unit test
 - b. A white-box test
 - c. An integration test
 - d. A functional test

Creating Unit Tests and Mock Services

Objectives

After completing this section, you should be able to implement unit tests and mock services to test applications.

Describing Unit Tests

Unit tests validate that specific pieces of code (units) meet the expected technical behavior. Units may be of any size, but in most programming languages unit tests tend to focus on a single function or method.

Unit tests are also referred to as *executable documentation* because they declare the expected behavior of code units while being also machine runnable.

Unit tests are in the base of the test pyramid. Those tests are intended to be the most numerous type of test and to be executed frequently.

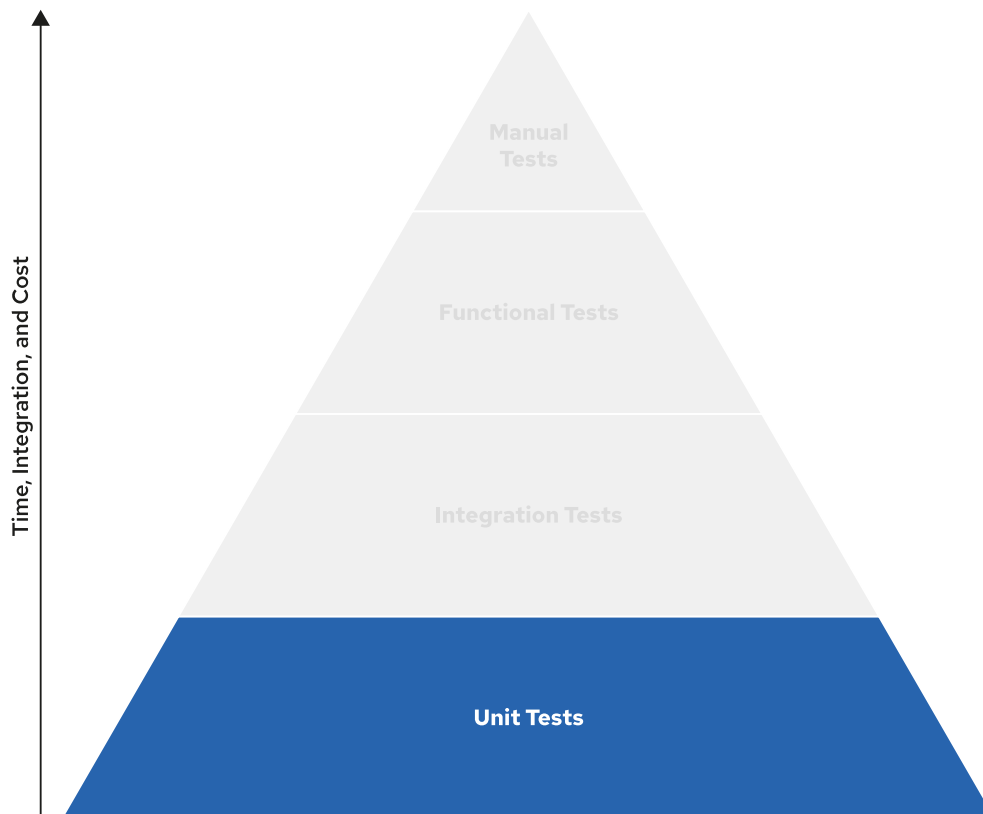


Figure 3.4: Unit Tests are the Base of the Test Pyramid

Despite the fact that there are no restrictions on implementation, unit tests should have the following properties:

Specific (or Small)

Unit tests should validate only one feature or capability. If a unit test verifies multiple capabilities then a test failure does not specify which one is failing.

Fast

If all unit tests execute in a short amount of time then developers can integrate the tests more easily in the development workflow. This leads to a safer development cycle and enables development approaches such as *test-driven development* or test integration into CI/CD pipelines. Developers are prone to disable slow tests from the development workflow, lowering the application's test coverage and risking bugs being introduced.

Isolated (or Trustworthy)

Unit tests should be independent of each other and from the execution environment. That is, the result of a unit test must be independent of other unit test executions, their outcome, or the environment they are running in.

Stateless

Unit tests must not rely on inputs generated by an external system, such as a database, the file system or a process. Each unit test must provide all values required by the test.

Idempotent (or Repeatable)

Results for the unit test must be independent from previous execution for this same test.

Structuring Unit Tests

Like any other kind of tests, unit tests can use the **Given-When-Then** notation.

The structure preparation section (**Given**) defines the variables, instances and values used in the unit. This section can also define **expectations**, which are values provided by dependencies of the unit under test.

The execution section (**When**) invokes the code unit under test. This invocation is a simple method call but can show other forms such as a process execution or a remote procedure call.

The results validation section (**Then**) asserts that the code unit meets the expected conditions. Expected conditions are based on values such as return values, shared status, result codes or exception thrown, but also on expected behaviours such as the number of method calls or process time.

```
public void timeout not exceeded testing() {
    val fibonacciCalculator = new FibonacciCalculator()
    val result = fibonacciCalculator.calculate(14)
    assertEquals(377, result)
}
```

- ❶ Test shows as functions invoked by the testing framework
- ❷ The **Given** section declares the instance of the `FibonacciCalculator` class needed to hold the `calculate` unit method.
- ❸ The **When** phase invokes the `calculate` method and retrieves the results.
- ❹ The **Then** section asserts the result is as expected.

Implementing Unit Tests

Unit tests are usually part of the application code. This allows the test a tight integration with the code base and enables white-box testing.

Test frameworks improve the expressiveness of tests making them clearer to the reader, but also enable unit tests as runnable elements. Different test frameworks for different programming languages provide different capabilities and usage methods.

The following sample unit test demonstrates how to implement a simple unit test in different languages and frameworks. It validates that the `FibonacciCalculator.calculate` method always returns 377 when invoked with 14 as a parameter. Note that we included a method running before each test, which instantiates the `fibonacciCalculator` variable:

Java

- JUnit

The *de facto* standard for unit tests in Java.

```
FibonacciCalculator fibonacciCalculator;

@BeforeEach
public void runBeforeEachTest() {
    fibonacciCalculator = new FibonacciCalculator();
}

@Test
public void timeout_not_exceeded() {
    assertEquals(377, fibonacciCalculator.calculate(14))
}
```

- TestNG

Inspired by **JUnit**, **TestNG** provides almost the same features. The only remarkable difference is that **TestNG** externalizes test aggregation in suites and parameterizes tests to XML files.

```
FibonacciCalculator fibonacciCalculator;

@BeforeMethod
public void runBeforeEachTest() {
    fibonacciCalculator = new FibonacciCalculator();
}

@Test
public void calculate_expected_value() {
    assertEquals(377, fibonacciCalculator.calculate(14))
}
```

Python

- unittest and unittest2

Integrated into the Python language, `unittest` provides the basic components for unit testing.

```

fibonacciCalculator: FibonacciCalculator

def setUp(self):
    fibonacciCalculator = FibonacciCalculator()

def calculate_expected_value(self):
    self.assertEqual(377, fibonacciCalculator.calculate(14))

```

- Pytest

Pytest extends `unittest` with more expressive semantics and more comprehensive reporting.

```

fibonacciCalculator: FibonacciCalculator

@pytest.fixture
def prepare():
    fibonacciCalculator = FibonacciCalculator()

def calculate_expected_value():
    assert 377 == fibonacciCalculator.calculate(14)

```

- Nose2

Self-proclaimed *unittest with plugins*.

```

class TestFibonacci(unittest.TestCase):

    fibonacciCalculator: FibonacciCalculator

    def prepare(self):
        fibonacciCalculator = FibonacciCalculator()

    @with_setup(prepare)
    def test():
        assert 377 == fibonacciCalculator.calculate(14)

```

Node.js

- Mocha

A feature-rich unit testing framework focused on running asynchronous tests.

```

var assert = require('assert');
describe('TestFibonacci', function() {

    var fibonacciCalculator;

    beforeEach(function() {
        fibonacciCalculator = new FibonacciCalculator();
    });

    describe('#calculate_expected_value()', function() {
        it('Calculator return 377 for fibonacci(14)', function() {

```

```
    assert.equal(377, fibonacciCalculator.calculate(14));
  });
});
});
```

- Jest

As opposed to Mocha, Jest targets for simplicity and interoperability with other JavaScript frameworks.

```
var fibonacciCalculator;

beforeEach(() => {
  fibonacciCalculator = new FibonacciCalculator();
})

test('Calculator return 377 for fibonacci(14)', () => {
  expect(fibonacciCalculator.calculate(14)).toBe(377);
});
```

Creating Test Doubles

One of the main complexities when developing unit tests is handling dependencies of the unit under test. Dependencies are other code units or services used by the unit under test, which are not part of the unit but can be seen as inputs for the unit.

Everything needed by the unit under test must be provided by the test, so the test must therefore provide the dependencies. This approach not only keeps the unit test specific and isolated, but also gives the developer options to make the test faster and easier to understand and maintain.

One way for the unit test to be independent of the dependencies is to use **test doubles**. A **test double** is a replacement of the dependencies by units with outcomes and behavior controlled by the test.

We can define several categories of **test doubles** depending on the way to prepare and use them:

Dummies

Dummies are dependencies that the test needs while creating or invoking the unit under test, but that are never used by the unit. Common examples are `null` values passed to a class constructor or to the method under test, or objects passed to constructor methods that are mandatory but never used during the test.



Fakes replace unit dependencies with a different implementation, much simpler and easier to manage during the test phase, but not suitable for production usage. Examples of fakes are in-memory databases or allow-all authenticators.



Stubs provide the code unit under test with canned responses. Use `stubs` when testing for expected behavior and responses of the unit against well-known dependency responses.



Mocks

Mocks are dependencies pre-programmed to respond to a limited set of requests. The unit test declares some **expectations** on how the dependency will be used by the unit and the responses it will provide. Use **mocks** when the original dependencies are complex to stub but easy to record.

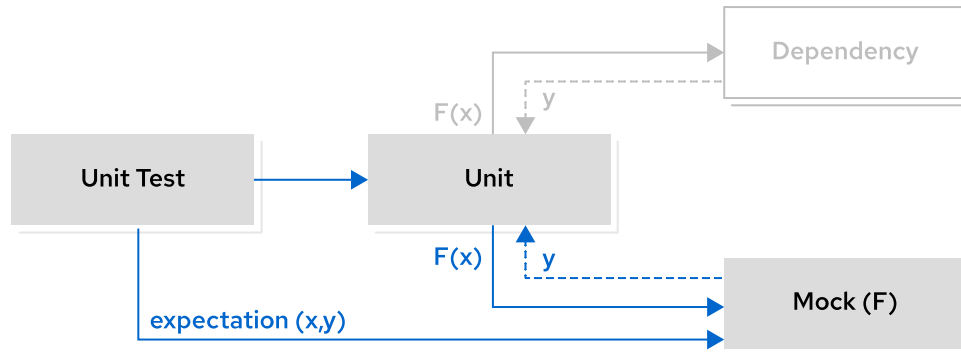


Figure 3.8: Mock type example

Spies

If a unit test needs not only to validate the outcome of a code unit but also the way it uses its dependencies then the unit test can use **spies**. **Spies** are **stubs** or **mocks** that record interactions with the unit under test. Unit test can then make assertions over the interactions.

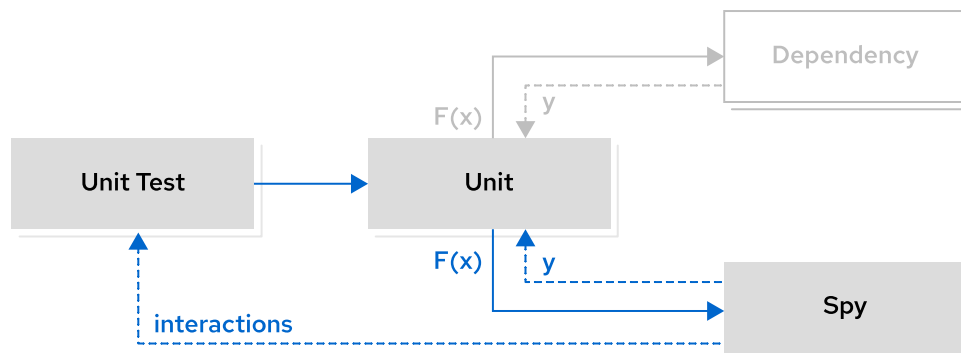


Figure 3.9: Spy type example

Multiple frameworks are available to help the developer create different kind of test doubles: **Mockito** and **EasyMock** in Java, **unittest.mock** in Python, or **Sinon.JS** in Node. Context Dependency Injection (CDI) frameworks are a powerful tool to create test doubles because they can declaratively create dependencies and inject them into tested units.



References

Wikipedia: Unit Testing

https://en.wikipedia.org/wiki/Unit_testing

Meszaros, Gerard. xUnit test patterns: Refactoring test code. Pearson Education, 2007.

Martin Fowler,

Mocks Aren't Stubs

<https://martinfowler.com/articles/mocksArentStubs.html>

Martin Fowler,

TestDouble

<https://martinfowler.com/bliki/TestDouble.html>

► Guided Exercise

Creating Unit Tests and Mock Services

In this exercise you will learn how to create unit tests and use test doubles to isolate these tests.

You will use the `school-library` application, which allows students to checkout books from a school library. You will unit test the `BookStats` class and the `Library` class included in the application.

The `school-library` application uses Quarkus [<https://quarkus.io/>], a developer-centric, cloud-native Java framework, which eases the development, testing, and deployment of applications. You will use Maven [<https://maven.apache.org/>], a well-known build automation tool used in Java projects, to run unit tests.

You will find the solution files for this exercise in the `solutions` branch of the `D0400-apps` repository, within the `school-library` folder.

Outcomes

You should be able to write a unit test that will test a specific part of the code in isolation, and isolate the unit being tested by using test doubles.

Before You Begin

To perform this exercise, ensure you have Git and the Java Development Kit (JDK) installed on your computer. You also need access to a command line terminal and your `D0400-apps` fork cloned in your workstation.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start every guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the `school-library` application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps/school-library` application folder and checkout the `main` branch of the `D0400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps/school-library
[user@host school-library]$ git checkout main
...output omitted...
```

- 1.2. Create a new branch to save any changes you make during this exercise.

```
[user@host school-library]$ git checkout -b unit-testing
Switched to a new branch 'unit-testing'
```

- ▶ 2. The `school-library` application can extract the number of words in a book. This behavior is encapsulated in the `BookStats` class. Create a unit test to verify that the `BookStats.countWords()` method correctly counts the number of words in a book.
 - 2.1. By using your editor of choice, review the code in the `src/main/java/com/redhat/training/BookStats.java` file. You will write a unit test for the `countWords` method contained in this file.
 - 2.2. Review the unit test file in the `src/test/java/com/redhat/training/BookStatsTest.java` file. Notice that there are already two tests in this file. The first test, named `countingWordsOfEmptyBookReturnsZero`, is completely implemented and checks that the word counter returns zero when the book is empty.

```
@Test
public void countingWordsOfEmptyBookReturnsZero() {
    // Given
    Book book = new Book("someISBN");

    // When
    double wordCount = BookStats.countWords(book);

    // Then
    assertEquals(0, wordCount);
}
```

The `@Test` annotation marks this function as a test case. The test case creates a new book by passing a book identifier or ISBN. The test does not specify any text for the book, so the book is empty. Finally, the test verifies that the number of words in the empty book is zero.

The second test, named `countingWordsReturnsNumberOfWordsInBook`, is incomplete and contains a failing assertion.

```
@Test
public void countingWordsReturnsNumberOfWordsInBook() {
    assertEquals(0, 1); // Replace this line with the actual test code...
}
```

- 2.3. Run the tests and check that the `countingWordsReturnsNumberOfWordsInBook` test fails. The `./mvnw test` command executes the maven test goal, which includes all tests found in the project and executes them.



Note

If you already have Maven 3.6.2+ installed in your system, you can use the `mvn` command instead of the `mvnw` wrapper script.

The Maven wrapper script is a way to run specific versions of Maven without having a system-wide installation of Maven.


```
[user@host school-library]$ ./mvnw test
...output omitted...
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.redhat.training.BookStatsTest
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.013 s
<<< FAILURE! - in com.redhat.training.BookStatsTest
[ERROR] countingWordsReturnsNumberOfWordsInBook Time elapsed: 0.01 s <<<
FAILURE!
org.opentest4j.AssertionFailedError: expected: <0> but was: <1>
    at
    com.redhat.training.BookStatsTest.countingWordsReturnsNumberOfWordsInBook
        (BookStatsTest.java:25)
...output omitted...
[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   BookStatsTest.countingWordsReturnsNumberOfWordsInBook:25 expected: <0>
but was: <1>
[INFO]
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
...output omitted...
```

The output shows that there is an error in line 25 of the `BookStatsTest.java` file. The assertion fails because 0 is not equal to 1.

- 2.4. Write the code to make the `countingWordsReturnsNumberOfWordsInBook` test pass. Replace the `assertEquals(0, 1)` line with the correct test implementation.

```
...output omitted...
@Test
public void countingWordsReturnsNumberOfWordsInBook() {
    // Given
    Book book = new Book("someISBN", "this is the content");

    // When
    double wordCount = BookStats.countWords(book);

    // Then
    assertEquals(4, wordCount);
}
...output omitted...
```

Note how the test is organized in three stages:

- Given stage: Sets up the scenario with a book containing four words.

- **When stage:** Calls the production code to be tested.
- **Then stage:** Asserts that the number of counted words is equal to four.

**Note**

Try to give meaningful and readable names to your tests to reflect the purpose of what you are testing.

This exercise uses the `camelCase` naming convention, but you can use other conventions if they look more readable to you.

2.5. Run the tests again and check that the two tests pass.

```
[user@host school-library]$ ./mvnw test
...output omitted...
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.redhat.training.BookStatsTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- **3.** Test the `checkout` method in the `Library` class. Create a unit test to verify that, after checking out a book from the library, one less copy is available in the book inventory.

3.1. Review the production code in the `src/main/java/com/redhat/training/Library.java` file. You will write a unit test for the `checkout` method contained in this file.

```
public Book checkout(String studentId, String isbn) throws
    BookNotAvailableException {
    if (!inventory.isBookAvailable(isbn)) {
        throw new BookNotAvailableException(isbn);
    }

    Book book = inventory.withdraw(isbn);
    loans.markAsBorrowed(studentId, book);

    return book;
}
```

3.2. Open the `src/test/java/com/redhat/training/LibraryTest.java` file and observe its contents. The `LibraryTest` class encapsulates unit tests for the `Library` class.

Note how the `setUp` function creates a library that uses an in-memory book inventory.

```
@BeforeEach
public void setUp() {
    inventory = new InMemoryInventory();
    library = new Library(inventory);
}
```

The `@BeforeEach` annotation specifies that the `setUp` function must be called before running each test. Therefore, each test gets a new inventory and a new library, ensuring the independence from previously executed tests.

- 3.3. Create a new unit test to validate that the number of inventory copies of a specific book decreases after checking out the book. Add the code to create the new test to the `LibraryTest` class.

```
...output omitted...
@BeforeEach
public void setUp() {
    inventory = new InMemoryInventory();
    library = new Library(inventory);
}

@Test
public void checkingOutDecreasesNumberOfBookCopiesFromInventory()
    throws BookNotAvailableException {
    // Given
    inventory.add(new Book("book1"));
    inventory.add(new Book("book1"));

    // When
    library.checkOut("someStudentId", "book1");

    // Then
    assertEquals(1, inventory.countCopies("book1"));
}
...output omitted...
```

Notice how the test is organized:

- Given a scenario with two copies of the `book1` book.
- When a student checks out a copy of `book1`.
- Then the number of inventory copies of `book1` should be one.

- 3.4. Run the tests. There should be one test run in the `LibraryTest`. All tests should pass.

```
[user@host school-library]$ ./mvnw test
...output omitted...
[INFO] -----
[INFO]  T E S T S
[INFO] -----
```

```
[INFO] Running com.redhat.training.BookStatsTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO] Running com.redhat.training.LibraryTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

**Note**

The execution order of the test suites may differ. Test results must be independent from the execution order.

- ▶ 4. Test an edge case of the checkout feature. If there are no copies of a book left in the inventory, then a student should not be able to checkout the book. Create a unit test to verify that the `Library.checkOut` method fails if there are no copies of the book left in the inventory.
 - 4.1. In the same `LibraryTest.java` file, add a new test to assert that an exception is thrown when you try to checkout a book with no copies in the inventory. Add the code for the new test as a new public method of the `LibraryTest` class.

```

...output omitted...
    // Then
    assertEquals(1, inventory.countCopies("book1"));
}

@Test
public void checkingOutUnavailableBookThrowsException()
    throws BookNotAvailableException {
    // Given
    inventory.add(new Book("book1"));
    inventory.add(new Book("book1"));

    library.checkOut("student1", "book1");
    library.checkOut("student2", "book1");

    // When
    final BookNotAvailableException exception = assertThrows(
        BookNotAvailableException.class,
        () -> {
            library.checkOut("student3", "book1");
        }
    );

    // Then
    assertTrue(exception.getMessage().matches("Book book1 is not available"));
}
...output omitted...

```

The preceding test does the following:

- The **Given** stage prepares the test scenario by adding two copies of a book with ID `book1` to the inventory. In the same scenario, a student checks out the two copies of the same book, so no copies of `book1` are left in the inventory.
- The **When** stage expects the `BookNotAvailableException` to be thrown by the `checkout` function when trying to checkout the third copy of the same book.
- The **Then** stage asserts that the exception includes the expected error message.

- 4.2. Run the tests. You should see two tests run in the `LibraryTest` class. All tests should pass.

```

[user@host school-library]$ ./mvnw test
...output omitted...
[INFO] -----
[INFO]  T E S T S
[INFO] -----
[INFO] Running com.redhat.training.BookStatsTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO] Running com.redhat.training.LibraryTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO]
[INFO] Results:
[INFO]

```

```
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

**Note**

Test your edge cases.

Edge cases normally define the boundaries of your software. These cases require special attention and are more prone to errors. Testing edge cases improves your protection against undesired behaviors and helps you find the limits of your application.

- 5. Check how tests fail when the production code produces unexpected results.

- 5.1. In the `src/main/java/com/redhat/training/Library.java` file, remove the condition that checks the inventory availability.

```
public Book checkOut(String studentId, String isbn)
    throws BookNotAvailableException {
    // Remove this block
    if (!inventory.isBookAvailable(isbn)) {
        throw new BookNotAvailableException(isbn);
    }

    Book book = inventory.withdraw(isbn);
    loans.markAsBorrowed(studentId, book);

    return book;
}
```

Verify that the `checkOut` method looks like the following:

```
public Book checkOut(String studentId, String isbn)
    throws BookNotAvailableException {
    Book book = inventory.withdraw(isbn);
    loans.markAsBorrowed(studentId, book);

    return book;
}
```

- 5.2. Run the tests again. The test named `checkingOutUnavailableBookThrowsException` should fail now.

```
[user@host school-library]$ ./mvnw test
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.redhat.training.BookStatsTest
```

```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.017 s -
in com.redhat.training.BookStatsTest
[INFO] Running com.redhat.training.LibraryTest
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.001 s
<<< FAILURE! - in com.redhat.training.LibraryTest
[ERROR] checkingOutUnavailableBookThrowsException Time elapsed: 0.001 s <<<
FAILURE!
org.opentest4j.AssertionFailedError: Unexpected exception type thrown
==> expected: <com.redhat.training.books.BookNotAvailableException> but was:
<java.lang.IndexOutOfBoundsException>
    at
    com.redhat.training.LibraryTest.checkingOutUnavailableBookThrowsException(Library
Test.java:66)
Caused by: java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0
    at com.redhat.training.LibraryTest.lambda$0(LibraryTest.java:67)
    at
    com.redhat.training.LibraryTest.checkingOutUnavailableBookThrowsException(Library
Test.java:66)
...output omitted...
```

Instead of receiving the expected `BookNotAvailableException` error, the test is receiving the `IndexOutOfBoundsException` error. The inventory throws this exception when trying to withdraw a book copy that does not exist.

- 5.3. Restore the inventory check in the `src/main/java/com/redhat/training/Library.java` file. The restored `checkOut` method should look like the following:

```
public Book checkOut(String studentId, String isbn)
    throws BookNotAvailableException {
    if (!inventory.isBookAvailable(isbn)) {
        throw new BookNotAvailableException(isbn);
    }

    Book book = inventory.withdraw(isbn);
    loans.markAsBorrowed(studentId, book);

    return book;
}
```

- 5.4. Run the tests again. Tests should pass.

```
[user@host school-library]$ ./mvnw test
...output omitted...
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.redhat.training.BookStatsTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO] Running com.redhat.training.LibraryTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
...output omitted...
[INFO]
[INFO] Results:
[INFO]
```

```
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 6. Mock the books inventory. In previous tests, the library used an in-memory implementation of the inventory (`InMemoryBookInventory`). In this step, you will use a mock instead. This validates that you are calling inventory methods as expected.

- 6.1. Open the `src/main/java/com/redhat/training/Library.java`. Notice how `Library` depends on the `BookInventory` interface.

```
...output omitted...
public class Library {

    private final Inventory inventory;
    private final LoanRegistry loans = new LoanRegistry();

    public Library(Inventory inventory) {
        this.inventory = inventory;
    }
    ...output omitted...
```

By depending on an abstraction, the library is not coupled to the specific implementation of an inventory. Therefore, you can inject any object implementing the `Inventory` interface. For example, you could provide a mocked inventory instead.

- 6.2. Open the `src/test/java/com/redhat/training/LibraryWithMockedInventoryTest.java` file. Observe how the inventory instance is now a mock of the `Inventory` interface in the `setUp` method. The test runner creates a new mock before running each test, by using the `mock` function from the Mockito library.

```
...output omitted...
@BeforeEach
public void setUp() {
    inventory = mock(Inventory.class);
    library = new Library(inventory);
}
...output omitted...
```

After the `setUp` method creates the mock, each test can define the desired behavior of the mock.

- 6.3. In the same file, create a new test to check that the `inventory.withdraw` method is called when a student checks out a book.

Add the `checkingOutWithdrawsFromInventoryWhenBookIsAvailable` test to the `LibraryWithMockedInventoryTest` class.


```

...output omitted...
public void setUp() {
    inventory = mock(Inventory.class);
    library = new Library(inventory);
}

@Test
public void checkingOutWithdrawsFromInventoryWhenBookIsAvailable()
    throws BookNotAvailableException {
    // Given
    when(inventory.isBookAvailable("book1")).thenReturn(true);

    // When
    library.checkOut("student1", "book1");

    // Then
    verify(inventory).withdraw("book1");
}
...output omitted...

```

The preceding test does the following:

- The **Given** stage configures the mock to set up a scenario in which the `book1` book is available. Specify the return value of the `isBookAvailable` method as `true` when called with the `book1` parameter.
- The **When** stage calls the `checkOut` method.
- The **Then** stage verifies that the `withdraw` method has been called with the `book1` parameter.

- 6.4. Run the tests. Verify there is one passing test in the `LibraryWithMockedInventoryTest` class.

```

[user@host school-library]$ ./mvnw test
[INFO] -----
[INFO]  T E S T S
[INFO] -----
...output omitted...
[INFO] Running com.redhat.training.LibraryWithMockedInventoryTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO]
...output omitted...
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...

```

- 6.5. Add another test that asserts the `inventory.withdraw` method is not called if the requested book is unavailable.

```

...output omitted...
    // Then
    verify(inventory).withdraw("book1");
}

@Test
public void checkingOutDoesNotWithdrawFromInventoryWhenBookIsUnavailable()
    throws BookNotAvailableException {
    // Given
    when(inventory.isBookAvailable("book1")).thenReturn(false);

    // When
    try {
        library.checkOut("student1", "book1");
    } catch(BookNotAvailableException e) {}

    // Then
    verify(inventory, times(0)).withdraw("book1");
}
...output omitted...

```

The preceding test does the following:

- The **Given** stage configures the mock so that the `book1` book is not available. This causes the `checkOut` method to throw a `BookNotAvailableException`.
- The **When** stage calls the `checkOut` method and catches the exception.
- The **Then** stage verifies that the `withdraw` method has not been called. The `times` function specifies that the method should have been called zero times.

6.6. Run the tests. Verify that all tests pass.

```

[user@host school-library]$ ./mvnw test
[INFO] -----
[INFO]  T E S T S
[INFO] -----
...output omitted...
[INFO] Running com.redhat.training.LibraryWithMockedInventoryTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, ...output omitted...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...

```

- 7. Commit all the changes to the exercise branch.

```
[user@host school-library]$ git add .  
...output omitted...  
[user@host school-library]$ git commit -m "finish exercise"  
...output omitted...
```

This concludes the guided exercise.

Creating Integration Tests

Objectives

After completing this section, you should be able to implement back-end integration tests.

Integration Tests

As you have learned previously, Integration Testing is a type of testing, which verifies that various individual modules interact with each other in an expected way.

While unit tests are created with only the domain knowledge relevant to the unit of work to test, integration tests require a far broader scope of the business domain knowledge to test.

Integration tests could cover validations at different levels: from testing how two classes interact, to validating the correct communication of two microservices.

Integration tests are in the second level of the testing pyramid. This means that integration tests require more work to write and execute more slowly than unit tests. Because of this added complexity, you execute them less frequently than unit tests, but they still must run as part of the testing pipeline.

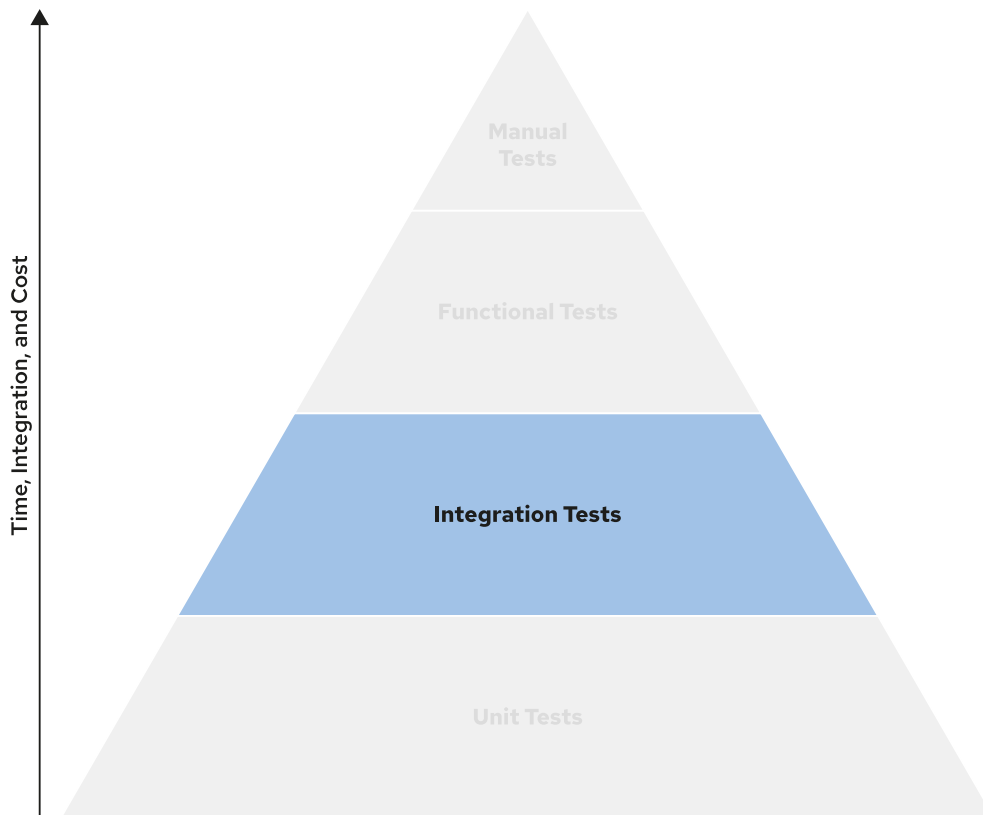


Figure 3.10: Integration Tests as the second level of the Test Pyramid

An integration test might be a *white-box test* or a *black-box test*, depending on how it is written and what it tests. For example, testing that a class communicates with another class could be considered a white-box integration test. An example of black-box integration testing would be a test that calls an API endpoint, which in turn saves the entity into the database. The test should verify that the response of the API method is HTTP response 201.

For more information about white-box versus black-box testing, see *Describing the Testing Pyramid*.

Differences Between Unit and Integration Tests

The line that separates unit tests and integration tests is not always clear. The following table can help you understand the differences between the two test types:

Differences Between Unit and Integration Tests

Unit Tests	Integration Tests
Test Business Logic in isolation	Test interaction between modules
Test all business rules in a module	Test expected output from dependent modules
Fake external dependencies	Use real dependencies
No setup	Setup external dependencies
Fast testing execution	Usually slower execution

Here is an example of an integration test written in Java:

```
@BeforeEach
public void setUp() {
    startAndSeedTestDatabase(); ❶
}

@AfterEach
public void teardown() {
    tearDownTestDatabase(); ❷
}

@Test
public void testRepositoryCanReadFromDB() {
    // Given
    val repository = new ProductRepository(); ❸

    // When
    val productList = repository.getAll(); ❹

    // Then
    assertThat(productList, is(not(empty()))); ❺
}
```

The structure of this test is:

- ❶ Starts the database and fills it with test data before each test.
- ❷ Stops the database after each test.
- ❸ Sets up the test prerequisites.
- ❹ Calls the method to test.
- ❺ Verifies that the output matches the expected value.

Integration Testing Methods

Depending on how you test these modules and how you treat the code inside said modules there are two approaches to testing: White-box and Black-box testing. Integration testing is no different in that regard to unit or functional testing.

White-box Testing

White-box integration testing takes into account the implementation details of each module, but it also assists in validating that the interfaces exposed by individual modules adhere to their contract and the interaction with dependent modules works as expected.

An example of white-box integration testing would be validating that the module under test calls another integrated module by using the expected methods and appropriate parameters.

Black-box Testing

Black-box integration testing validates a business requirement without the knowledge of which other components take part in the operation.

An example of black-box integration testing is creating a test that calls one of your API methods, which in turn saves the entity into the database. The test should verify that the response of the API method has a 201 HTTP response.

The difference between white box and black box testing isn't always clear. For example, consider that in the previous scenario apart from testing the API call result, you could also test that the actual resource is saved in the database. If you consider the database record as an output of the API call, then that test would also be black-box testing. But if you consider the database as an internal component, then it is now part of the implementation details and it could be considered white-box testing.

Component Testing Technique

The most basic integration test you can write is to verify that the communication with an external component, such as a database, works as expected.

A unit test might cover this feature by creating a test double to replace the database and then validating if a module operation saves an entity.

An integration test, on the contrary, would start up a test database as a fake external component, and make sure that the application code connects to it or that the application is resilient to an unexpected connection drop to the database.

Both approaches involve testing interactions with external components: unit test from the code perspective and integration tests from the component perspective. For this reason, integration tests are also referred to as *component tests*.

Approaches to Integration Testing

Depending on how you start and develop integration tests there are several approaches, each having its own advantages and disadvantages.

Big Bang

The first approach available while doing integration testing is the Big Bang. This approach treats all the units required for a business process as a single entity and tests it as such. Although it might have the advantage of testing the whole business process, it has two main disadvantages: The testing cannot begin until the development of all the units involved finishes, and that finding the point of failure when an assertion fails becomes difficult.

This approach tries to tackle all the business logic at once and usually becomes hard to develop and maintain, and takes a lot of time.

Incremental Testing

To avoid the complexity of Big Bang testing, a more incremental approach is preferred. Incremental testing takes the modules to be tested first in pairs and then gradually increases the number of modules involved in the testing process until all the relevant modules are covered.

During this incremental testing, if a module interacts with another module that is not available, you must replace it with a test double.

Depending on where this incremental testing begins and how it proceeds we can use several approaches to perform incremental testing.

Top-down

If we categorize modules from the ones closest to the user on top and the furthest ones at the bottom, the top-down approach starts testing the most critical interaction to the user and travels down to the least critical. You must replace any modules missing during the testing procedure with test doubles.

This approach has the advantage of finding the most critical errors earlier in the development process, but at the cost of a higher work effort to develop the needed test doubles.

If a lower module is not ready to use during the test, then it can be replaced by a dummy module. Those dummy modules responding to the module under test are named *stubs*.

Bottom-up

As opposed to the top-down approach, when you are using the bottom-up approach you start with lower modules, which are usually easier to test and require fewer test doubles. This creates a chain of confidence when going up in the dependency tree to test higher modules. This technique has the added value of quickly finding errors in bottom modules before those errors impact modules closer to the user. Dummy modules used to replace upper modules not ready during the test are named *drivers*.

Sandwich or Hybrid

The Sandwich or Hybrid approach takes its name from the result of combining the top-down and bottom-up approaches. This combination is achieved by testing the higher modules and then testing the lower modules at the time of integrating them with the higher ranking ones.

Hybrid integration test uses both *stubs* and *drivers*.

Hybrid integration testing is the preferred approach for big projects or projects having other projects as dependencies. As a rule of thumb, prefer bottom-up integration testing for green-field projects and top-down integration testing for brown-field projects.



References

Integration Test

<https://martinfowler.com/bliki/IntegrationTest.html>

White-box testing

https://en.wikipedia.org/wiki/White-box_testing

Black-box testing

https://en.wikipedia.org/wiki/Black-box_testing

► Guided Exercise

Creating Integration Tests

In this exercise you will add integration tests to the initial implementation of a shopping cart.

The Quarkus application consists of the following components:

- **CartService:** the main shopping cart functionality with its own logic.
- **CatalogService:** a simulation of an external service that provides the available products.
- Auxiliary classes and files to make the different services work together.

The shopping cart application exposes an API and includes an OpenAPI endpoint.

You will find the solution files for this exercise in the `solutions` branch of the `D0400-apps` repository, within the `shopping-cart` folder.



Note

This guided exercise only covers a portion of all the tests you must create for the shopping cart application. Use coverage techniques to identify all the areas of your application that you must test.

Outcomes

You should be able to create integration tests following white-box and black-box techniques.

Before You Begin

To perform this exercise, ensure you have Git and the Java Development Kit (JDK) installed on your computer. You also need access to a command line terminal and your `D0400-apps` fork cloned in your workstation.

Use a text editor like `VSCodium`, which supports syntax highlighting for editing source files.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the `shopping-cart` application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps/shopping-cart` application folder and checkout the `main` branch of the `D0400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps/shopping-cart
[user@host shopping-cart]$ git checkout main
...output omitted...
```

- 1.2. Create a new branch to save any changes you make during this exercise.

```
[user@host shopping-cart]$ git checkout -b integration-testing
Switched to a new branch 'integration-testing'
```

- ▶ 2. Create an integration test following the white-box method to cover the process of adding a product to the cart.

With the white-box method the tests are focused on the implementation details of the application.

- 2.1. Open the `src/main/java/com/redhat/shopping/cart/CartService.java` file. Inspect the code for the `addProduct` method and the use of the catalog service.

The `addProduct` method checks if the product you are trying to add to the cart is already in your list of products. If this is not the case, then the cart service adds the product to your list of products and updates the cart totals. The cart service includes the `totalItems` method, which returns the total number of items in the cart.

- 2.2. Open the `src/main/java/com/redhat/shopping/catalog/CatalogService.java` file and examine the code.

The catalog service implementation initializes a fixed list of products. Only products with an ID between 1 and 5 are available in the catalog service. When a non existing product is requested a `ProductNotFoundInCatalogException` exception is raised.

- 2.3. Create a test file `src/test/java/com/redhat/shopping/integration/whitebox/ShoppingCartTest.java` with the following content:

```
package com.redhat.shopping.integration.whitebox;

import com.redhat.shopping.cart.CartService;
import com.redhat.shopping.catalog.ProductNotFoundInCatalogException;
import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import javax.inject.Inject;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

@QuarkusTest
public class ShoppingCartTest {

    @Inject
    CartService cartService;

    @BeforeEach
    void clearCart() {
```

```

        this.cartService.clear();
    }

    // @todo: add integration tests
}

```

The preceding code injects the service to be tested and cleans the cart content before each test is executed.

- 2.4. Create a test for the scenario of a user adding a non existing product in the catalog to the cart. The specification for the test is:

- Given an empty cart, a product that does not exist in the catalog and a quantity.
- When the product is added to the cart.
- Then a `ProductNotFoundInCatalogException` exception is raised.

Update the `src/test/java/com/redhat/shopping/integration/whitebox/ShoppingCartTest.java` file and append the test implementation:

```

@Test
void addingNonExistingProductInCatalogRaisesAnException() {
    assertThrows(
        ProductNotFoundInCatalogException.class,
        () -> this.cartService.addProduct(9999, 10)
    );
}

```

- 2.5. Run the tests and verify that the tests pass.

```

[user@host shopping-cart]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...

```

- 2.6. Create a test for the scenario of a user adding a new product to an empty cart. The specification for the test is:

- Given an empty cart and an existing product that is not in the cart.
- When the product is added to the cart 10 times.
- Then the product is in the cart, and the total of items in the cart is equal to 10.

Update the `src/test/java/com/redhat/shopping/integration/whitebox/ShoppingCartTest.java` file and append the test implementation:

```
@Test
void addingNonExistingProductInCartTheTotalItemsMatchTheInitialQuantity()
    throws ProductNotFoundInCatalogException {

    this.cartService.addProduct(1, 10);

    assertEquals(10, this.cartService.totalItems());
}
```

2.7. Run the tests and verify that the two tests pass.

```
[user@host shopping-cart]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

2.8. Create a test for the scenario of a user adding more items of a product that is already in the cart. The specification for the test is as follows:

- Given an empty cart, an existing product that is already in the cart.
- When the product is added 100 more times to the cart.
- Then the total of items in the cart is the sum of the existing quantity and the initial quantity of the product.

Update the `src/test/java/com/redhat/shopping/integration/whitebox/ShoppingCartTest.java` file and append the test implementation:

```
@Test
void addingProductThatIsInTheCartTheTotalItemsMatchTheSumOfQuantities()
    throws ProductNotFoundInCatalogException {

    this.cartService.addProduct(1, 10);
    this.cartService.addProduct(1, 100);

    assertEquals(110, this.cartService.totalItems());
}
```

▶ 3. Run the tests and verify that they all run without any failures or errors.

```
[user@host shopping-cart]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

▶ 4. Create an integration test following the black-box method to cover the process of removing a product from the cart.

With the black-box method you write the tests without knowing the internal structure of the application.

- 4.1. Start the shopping cart application.

```
[user@host shopping-cart]$ ./mvnw quarkus:dev
...output omitted...
Listening for transport dt_socket at address: 5005
...output omitted...
...output omitted... Listening on: http://localhost:8080
...output omitted...
```

- 4.2. Navigate to `http://localhost:8080/swagger-ui` by using a web browser to see the API documentation.
- 4.3. Click **DELETE /cart/products/productId** to examine the documentation for the endpoint, which implements the removal of products from the cart.
The endpoint expects a parameter with the product ID, and the response codes can be one of the following:
 - 200: when the product was successfully removed from the cart.
 - 204: when the product was successfully removed from the cart and now the cart is empty.
 - 400: when the product is not in the catalog.
 - 404: when the product is not in the cart.

This information is also present as `@APIResponse` annotations on the `removeFromCart` method for the `com.redhat.shopping.ShoppingCartResource` class.

- 4.4. Terminate the application by pressing `Ctrl+C` in the terminal window.
- 4.5. Create a test file `src/test/java/com/redhat/shopping/integration/blackbox/ShoppingCartTest.java` with the following content:

```
package com.redhat.shopping.integration.blackbox;

import com.redhat.shopping.cart.AddToCartCommand;
import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.Random;

import static io.restassured.RestAssured.delete;
import static io.restassured.RestAssured.given;

@QuarkusTest
public class ShoppingCartTest {

    private int randomQuantity() {
        return (new Random()).nextInt(10) + 1;
    }
}
```

```

    }

    private void addProductToTheCartWithIdAndRandomQuantity(int productId) {
        AddToCartCommand productToAdd = new AddToCartCommand(
            productId,
            this.randomQuantity()
        );

        given()
            .contentType("application/json")
            .body(productToAdd)
            .put("/cart");
    }

    @BeforeEach
    public void clearCart() {
        delete("/cart");
    }

    // @todo: add integration tests
}

```

The preceding code cleans the cart content before each test is executed and defines some helper methods to use in the tests.

- 4.6. Add a test to cover the process of removing a non existing product in the catalog from the cart. The specification for the test is:

- Given a non existing product in the catalog.
- When the product is removed from the cart.
- Then the status code of the response is 400.

Update the `src/test/java/com/redhat/shopping/integration/blackbox/ShoppingCartTest.java` file and append the test implementation:

```

@Test
public void removingNonExistingProductInCatalogReturns400() {
    given()
        .pathParam("id", 9999)
    .when()
        .delete("/cart/products/{id}")
    .then()
        .statusCode(400);
}

```

The preceding test is agnostic about the implementation details of the process of removing a non existing product in the catalog from the cart. The test only uses the described input and output available in the API documentation to verify the integration of the different components of the application.

Internally, the testing framework runs the application in the background, sends the ID 9999 to the `delete` endpoint, and checks that the response code is 400.

The direct usage of application programming interfaces (APIs) as part of integration testing is also known as API testing.

- 4.7. Add a test to cover the process of removing a product from the cart when the product is not already in the cart. The specification for the test is:

- Given an existing product that is not in the cart.
- When the product is removed from the cart.
- Then the status code of the response is 404.

Update the `src/test/java/com/redhat/shopping/integration/blackbox/ShoppingCartTest.java` file and append the test implementation:

```
@Test
public void removingNonAddedProductToTheCartReturns404() {
    given()
        .pathParam("id", 1)
    .when()
        .delete("/cart/products/{id}")
    .then()
        .statusCode(404);
}
```

- 4.8. Add a test to cover the process of removing a product from the cart when the product is already in the cart and is the only one stored. The specification for the test is:

- Given an existing product that is the only one in the shopping cart.
- When the product is removed from the cart.
- Then the status code of the response is 204.

```
@Test
public void removingTheOnlyProductInCartReturns204() {
    this.addProductToTheCartWithIdAndRandomQuantity(1);

    given()
        .pathParam("id", 1)
    .when()
        .delete("/cart/products/{id}")
    .then()
        .statusCode(204);
}
```

- 4.9. Add a test to cover the process of removing a product from the cart when the product is already in the cart and is not the only one stored. The specification for the test is:

- Given an existing product and a cart with other products in it.
- When the product is removed from the cart.
- Then the status code of the response is 200.

```

@Test
public void
    removingProductFromCartContainingMultipleAndDifferentProductsReturns200() {
    this.addProductToTheCartWithIdAndRandomQuantity(1);
    this.addProductToTheCartWithIdAndRandomQuantity(2);

    given()
        .pathParam("id", 1)
    .when()
        .delete("/cart/products/{id}")
    .then()
        .statusCode(200);
}

```

- 5. Run the tests and verify that they all run without any failures or errors.

```

[user@host shopping-cart]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...

```

- 6. Commit all the changes to the exercise branch.

```

[user@host shopping-cart]$ git add .
...output omitted...
[user@host shopping-cart]$ git commit -m "finish exercise"
...output omitted...

```

This concludes the guided exercise.

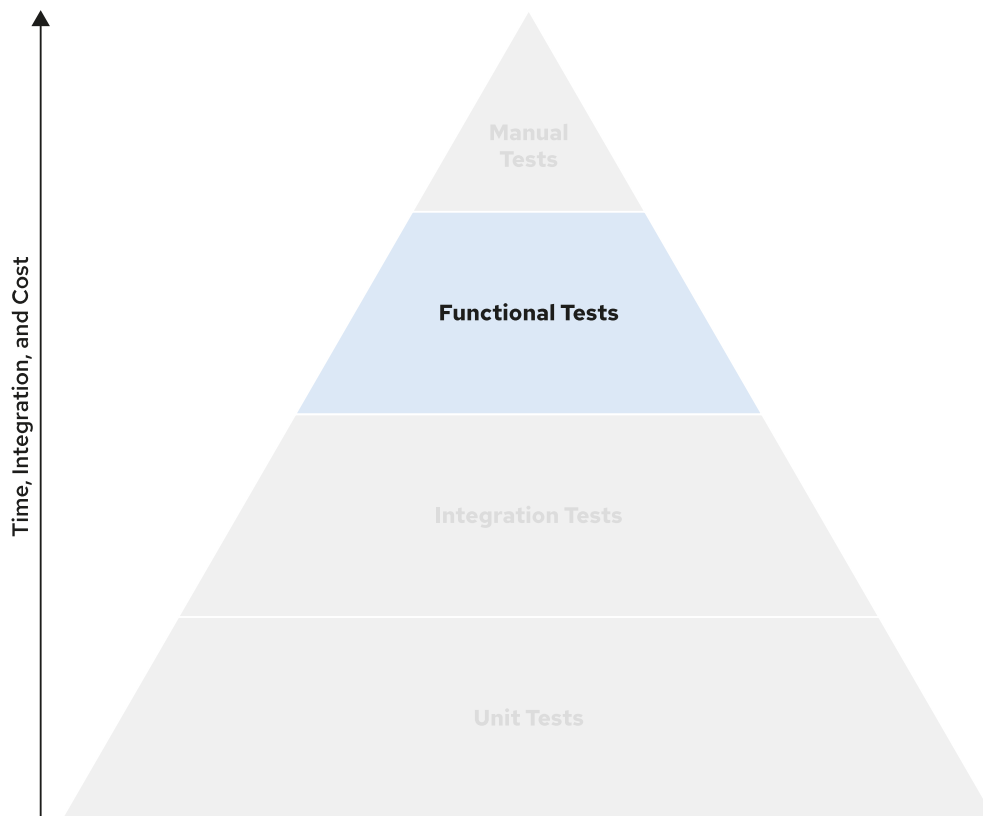
Building Functional Tests

Objectives

After completing this section, you should be able to create tests to validate that an application meets its functional requirements.

Functional Testing

In the testing pyramid, *functional tests* sit at the highest level of *automated tests*.



Although *integration tests* focus on the communication and *integration* between components, functional tests focus on meeting user scenarios and *functionality* requirements.

**Note**

The distinction between types of tests is not always obvious, especially as you go up the testing pyramid. What one team defines as functional tests might be deemed end-to-end or acceptance tests by another team. Gradually, each team will come to define their own particular boundaries.

For this course, we will always refer to automated tests for user-facing features as *functional tests*.

When writing functional tests, try to put yourself in the position of your users.

As you will see in the sections about *test-driven development* (TDD), writing your tests first and with your users in mind can greatly improve the design of your application before writing any application code.

Although previous sections focused mainly on *white-box tests*, which hook into the code itself to test specific pieces, functional tests are *black-box tests*. For more information about the difference between white-box and black-box testing, see *Describing the Testing Pyramid*.

Cost Versus Value of Functional Tests

Being so high up on the testing pyramid, you can expect functional tests to differ from lower tests in a number of ways.

Functional tests run in an environment closer to the real world than most other automated tests. Therefore, running them usually takes **more time and resources**. They are also more prone to breaking as they depend on more parts of the system and require more maintenance.

Because of these high costs, it is often ideal to only test the bare minimum of what is *most valuable* for your team. This might be the entire application or just a subset of functionality, which has the highest value to the user or business.

Simultaneously, do not underappreciate functional tests. The exercise of developing functional tests helps the entire team, especially developers, to think about the application from the user's perspective. It is often easy for teams to become so focused on *how* to build features that they forget *why* it is valuable in the first place.

Beyond user-centric value, functional tests are often the best way for developers to communicate with stakeholders on the specifics of how an application should run. Also, they clearly demonstrate whether the current implementation matches the agreed specification, from a high level.

Behavior-driven Development

In fact, there is a form of development centered around automating functional tests that implement an agreed-upon specification, named *behavior-driven development* (BDD). Based on the ideals of *test-driven development* (TDD), which is covered in a separate section, BDD codifies the following workflow:

1. Developers and stakeholders jointly write a specification on how the application should work. This specification often follows the *given-when-then* format and outlines functionality and conditions for the application in plain language. An example of the given-when-then format is provided later in this section.
2. Developers use this specification directly to create a functional test suite that asserts the specification is implemented and working.

3. Developers implement changes in the application that make the tests pass.

Note that, like with TDD, developers write the test cases *first*. Writing the test cases before writing the code helps solidify and confirm how the application should work.

In many cases, the discussions around forming the initial specification lead to improvements and error avoidance before any application code is written. Correcting issues this early in the software development lifecycle is significantly cheaper and quicker than discovering them later.

Example of given-when-then syntax:

```
GIVEN the user is logged in
AND the user has the MANAGE-BOOKS permission
WHEN the user clicks "Delete Book" next to the book named ""
AND confirms their selection
THEN the book should be deleted
```

Often, a syntax named *Gherkin* can help the developer map this given-when-then language to specific test cases and steps. This tooling is optional to practicing BDD and exact tool choice depends on the application and testing architecture.

Implementing Functional Tests

Often, you use a library or framework to actually write functional tests. Which library fits best depends on the application software architecture.

Note that this dependency is less strict than it is with white box tests. For example, JUnit can only unit test Java code, but most browser-based interface testing frameworks will work regardless of how the pages are built.

User Interface (UI) Testing

For brevity, we will focus solely on browser-based testing frameworks. These are currently the most common type of interfaces amongst enterprise software applications.

Although there are *many* browser testing frameworks available, some of the more popular ones are:

- Selenium, the oldest and most popular browser testing framework.
- Cypress, a JavaScript testing library, which runs in the same various browsers as your application UI. The main goal behind Cypress is to alleviate some of the frustrations when using testing tools that run outside the browser environment, especially when testing SPAs.
- Puppeteer/Playwright, relatively new frameworks, which aim to execute even closer to your application code than Cypress. Being so new, there are still occasional pain points around initial set up.

Compared to the others, Selenium is an older framework. As such, it has undergone substantial change as web development practices have changed. The most notable change being a shift to developing *single-page applications* (SPAs).

**Note**

A single-page application or SPA fundamentally alters how the browser loads the web page or web application. Instead of downloading a fully-rendered page from the server and then reloading an entire new page on user interaction, an SPA dynamically loads parts of the page, called *components*, as they are needed.

Tests and test frameworks cannot make the assumption that the entire page is ready to be tested on page load. The benefit is that some frameworks can allow the developer to run tests on individual components.

Cypress, Puppeteer, and Playwright were all developed after the advent of SPAs. They were partly developed as a means to simplify testing workflows around SPAs.

Running Cypress

Installing Cypress requires Node.js 10 or higher to be installed. You will also need a simple Node.js project in order to save Cypress as a dependency.

A sample Node.js application is also provided as part of this course, located at <https://github.com/RedHatTraining/DO400-apps/tree/main/greeting-service>. You will use the Scoreboard Node.js project in that repository as part of the accompanying guided exercise, where you will also write and run a set of functional tests by using Cypress.

You may also initialize a new Node.js project by creating a new project directory, running `npm init` within the directory, and following the prompts.

Install Cypress with NPM and save it as a dependency:

```
[user@host greeting-service]$ npm install cypress --save-dev
...output omitted...
added 365 packages, and audited 365 packages in 53s
...output omitted...
```

Run Cypress with NPX, which is included as part of an NPM installation:

```
[user@host greeting-service]$ npx cypress open
It looks like this is your first time using Cypress: 6.3.0
...output omitted...
Opening Cypress...
```

This will open the Cypress UI where you can manually trigger the included default test scripts.

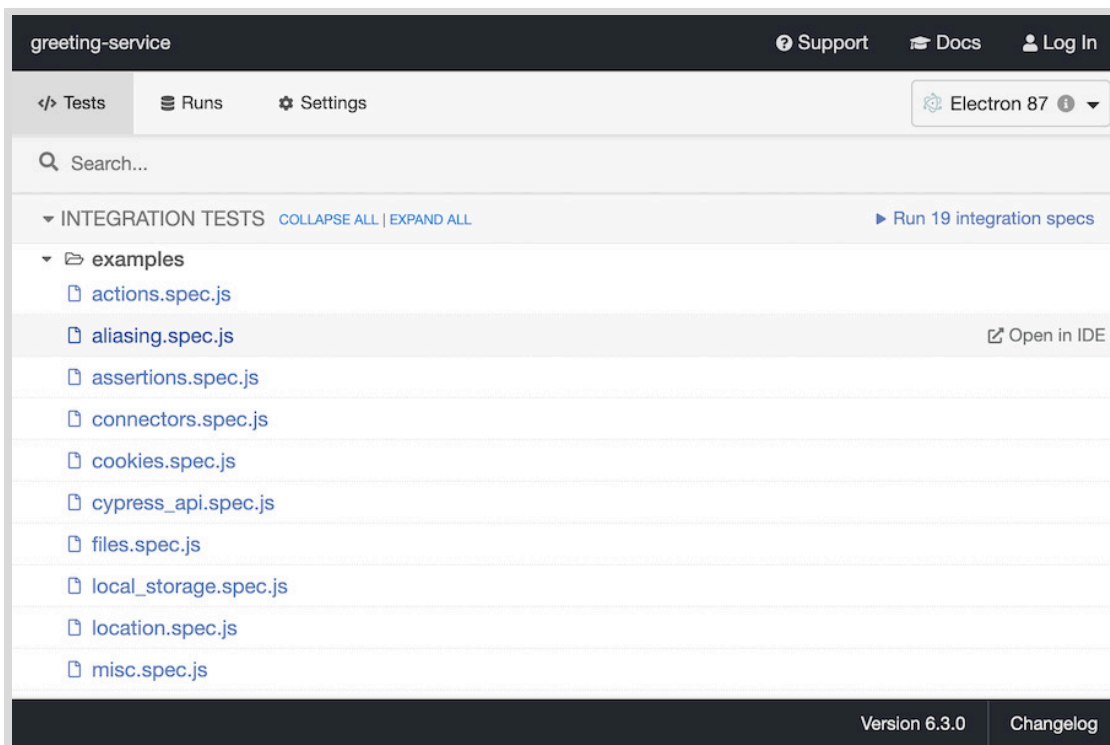


Figure 3.12: Cypress testing dialog

Note that you can select a browser for Cypress to use with the selection menu at the top right of the window.

Run a test by clicking on the file name. For example, clicking on `actions.spec.js` will run all of the tests in `greeting-service/cypress/integration/examples/actions.spec.js`.

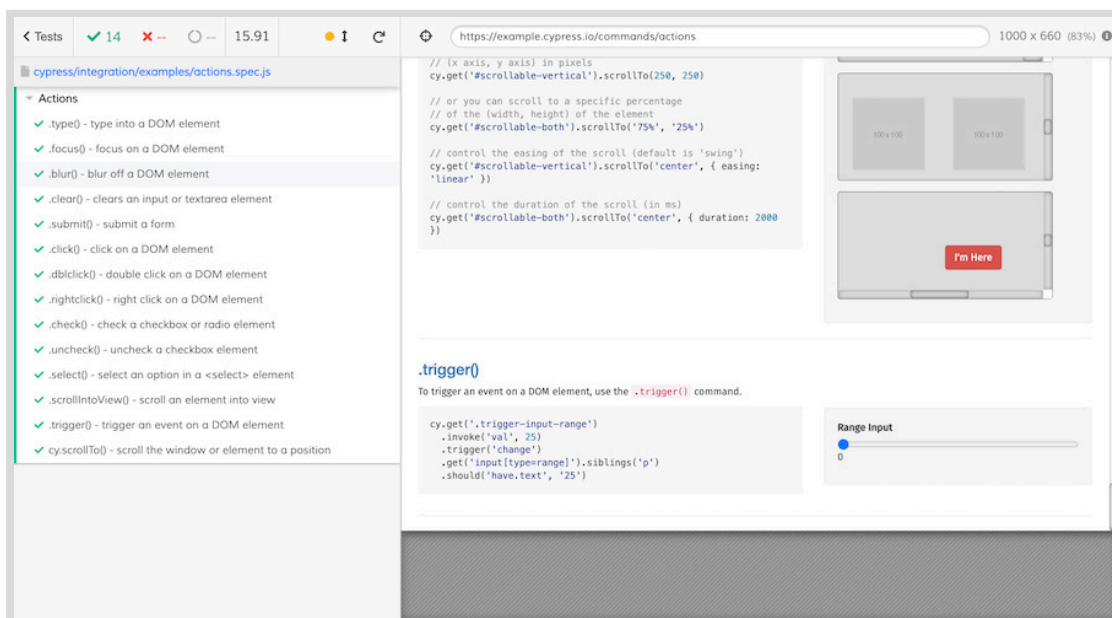


Figure 3.13: Example Cypress test execution

These examples are created by Cypress the first time you run it, if you do not have any tests already configured. You can create additional test files inside the `cypress` directory, which was created by this initial execution.

Close all Cypress windows when you are finished.

API Testing

Especially for testing purposes, consumers of your *application programming interface* (API) can also be considered your users.

In order to break down your API into separate pieces for functional testing, start by focusing on a single suite of API endpoints.

For example, your application exposes the following API route:

`https://example.com/books/add`

To add a book, your system calls several files, methods, classes, etc. The unit tests might have a test per method.



Note

This example assumes an object-oriented programming model. For other programming styles, the overall pattern would be similar.

However, a suite of *functional tests* might cover *all* of the functionality for *books*, with a few tests specifically covering *add*. An individual functional test would *not* directly call one of the underlying class methods. The test case would most likely invoke the functionality by calling the API *add* endpoint.

Compared to UI testing, choosing an API testing framework is more dependent on your application's programming language and library choice. These days, most APIs are based around HTTP and REST, which serve to standardize programming interfaces. This standardization eases testing library design and development.

Although this course does not focus on API testing specifically, some popular open source options for testing REST-based HTTP APIs include:

- Insomnia, an API testing tool that includes a developer UI for making HTTP requests.
- Rest-Assured, a Java domain-specific language (DSL) that provides given-when-then syntax and shortcuts for testing APIs built in Java.
- SoapUI, one of the most popular API testing tools. It supports testing both REST and SOAP web services, among others.



References

The Practical Test Pyramid

<https://martinfowler.com/articles/practical-test-pyramid.html#acceptance>

Simple Programmer Automated Testing Guide

<https://simpleprogrammer.com/ultimate-automation-testing-guide/>

Selenium

<https://www.selenium.dev>

Puppeteer

<https://developers.google.com/web/tools/puppeteer/>

Playwright

<https://playwright.dev>

Cypress

<https://www.cypress.io>

Installing Cypress

<https://docs.cypress.io/guides/getting-started/installing-cypress.html#System-requirements>

Insomnia

<https://github.com/Kong/insomnia>

Rest-Assured

<https://github.com/rest-assured/rest-assured>

► Guided Exercise

Building Functional Tests

In this exercise you will execute, fix, and add functional tests to an example application by using the Cypress testing framework.

Cypress is written in Node.js, the well-known JavaScript runtime. To install and use Cypress, you must use the Node.js package manager (npm).

You will find the solution files for this exercise in the `solutions` branch of the `D0400-apps` repository, within the `scoreboard` folder.

Outcomes

You should be able to write browser-based functional tests.

Before You Begin

To perform this exercise, ensure you have the following:

- Git
- Node.js ≥ 12
- Chrome ≥ 67 or Firefox ≥ 60
- Your `D0400-apps` fork cloned in your workstation

Optional: in order to run Cypress in a container, make sure you have Podman or Docker installed.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the scoreboard application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps/scoreboard` application folder and checkout the `main` branch of the `D0400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps/scoreboard
[user@host scoreboard]$ git checkout main
...output omitted...
```

- 1.2. Create a new branch to save any changes you make during this exercise.


```
[user@host scoreboard]$ git checkout -b functional-testing
Switched to a new branch 'functional-testing'
```

- ▶ 2. Run the Scoreboard example application. This is the application that you are functionally testing.

2.1. Install dependencies with `npm install`. These dependencies include Cypress.

```
[user@host scoreboard]$ npm install
...output omitted...
Cypress 5.4.0 is installed in ...output omitted.../Cypress/5.4.0

added 1743 packages from 853 contributors and audited 1745 packages in 31.194s
...output omitted...
```



Note

The Node.js package manager (`npm`) is the default tool to manage dependencies and automate builds in Node.js projects. The `npm install` command downloads and installs the dependencies specified in the `package.json` file. You can use this file to define additional dependencies, configuration, and scripts at the project level.



Important

The Cypress installation process might be slow in certain systems.

2.2. Start the UI development server with `npm start`.

```
[user@host scoreboard]$ npm start
Compiled successfully!

You can now view scoreboard in the browser.

Local:           http://localhost:3000
On Your Network: http://LOCAL_NETWORK_IP:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

The development server can take a few minutes to start up. The output will only look like the preceding example once it is ready.

2.3. Once the development server has started, it should automatically open your web browser to the application. If not, open your browser to `http://localhost:3000`.



Important

Leave the development server running while executing the tests. It is necessary for the tests to succeed.

► 3. Run the functional tests without the graphical interface.

- 3.1. In a new terminal session, change directory to the `scoreboard` application again. Then, run the NPM script to start Cypress.

```
[user@host scoreboard]$ npm run cy:run
...output omitted...
      Spec                                Tests  Passing  Failing  Pending  Skipped
+-----+
#   scoreboard.spec.ts    00:01      3        2        -        1        -
+-----+
#   All specs passed!    00:01      3        2        -        1        -
```

The tests run *without* opening a browser. This mode is called "headless".



Important

Cypress verifies the installation when you run it for the first time. The verification process checks that your system has all the required libraries to run Cypress. If you use Linux, then you might need additional dependencies in your system. Refer to the Installing Cypress [<https://docs.cypress.io/guides/getting-started/installing-cypress/>] guide for more details.

If the verification fails due to a time-out, then you might try to run it again by using the `npx cypress verify` command.

► 4. Run the included functional tests with the graphical interface.

- 4.1. Run the `cy:open` script to open the Cypress development window.

```
[user@host scoreboard]$ npm run cy:open
...output omitted...
```

- 4.2. Select a browser at the top right of the window. The options shown will depend on which browsers you have installed.
Select the `Electron` option. This is the Chromium-based browser that comes with Cypress.

- 4.3. Click the `scoreboard.spec.ts` file name to run that test suite.
This will open the browser and run the tests. Two of the three tests should have a green check mark. The third test should be greyed out to indicate that it was skipped for now.

- 4.4. Close the browser and Cypress windows when finished.

► 5. Fix a broken functional test.

- 5.1. Open `cypress/integration/scoreboard.spec.ts` in your editor.
- 5.2. Change `it.skip` to just `it` for the test named `should decrease a player's score`. Adding `.skip` to a test tells Cypress to skip that particular test case.
The test case should look like the following:

```

...output omitted...
it("should decrease a player's score", () => {
  // AND tom has a score of 0
  cy.get("#player-scores").should("contain", "tom: 0");

  // WHEN the user hits '-' next to 'tom'
  cy.get("#player-scores")
    .contains("tom")
    .siblings()
    .contains("-")
    .click();

  // THEN 'tom's score should be -1
  cy.get("#player-scores").should("contain", "tom: 0");
});
...output omitted...

```

5.3. By using either `cy:open` or `cy:run`, run the test suite again.

The newly unskipped test fails the last assertion. Specifically, it displays the error expected `<div#player-scores>` to contain `tom: 0`.

5.4. Change the test to assert that Tom's new score is `-1`. The test case should look like the following:

```

...output omitted...
it("should decrease a player's score", () => {
  // AND tom has a score of 0
  cy.get("#player-scores").should("contain", "tom: 0");

  // WHEN the user hits '-' next to 'tom'
  cy.get("#player-scores")
    .contains("tom")
    .siblings()
    .contains("-")
    .click();

  // THEN 'tom's score should be -1
  cy.get("#player-scores").should("contain", "tom: -1");
});
...output omitted...

```

5.5. Run the test suite again. All of the tests should pass.

```
[user@host scoreboard]$ npm run cy:run
```

```
...output omitted...
```

Spec	Tests	Passing	Failing	Pending	Skipped
# scoreboard.spec.ts	00:01	3	3	-	-
# All specs passed!	00:01	3	3	-	-

**Note**

If you ran Cypress with `cy:open`, then you do not need to manually run the tests again. In its graphical mode, Cypress detects file changes and automatically runs the tests.

► **6.** Implement a new test scenario.

- 6.1. In `scoreboard.spec.ts`, add a new test scenario that validates a maximum player name length.

Add another case to the `add player` suite. Be sure to add the following within the `describe` function call alongside the other cases.

```
it("should only allow names of max length 10", () => {
  // AND the user has entered 10 characters into the 'Player Name' field
  const nameField = cy.get("form").find('[placeholder="Player Name"]');
  nameField.clear();
  nameField.type("1234567890");

  // WHEN the user enters an 11th character
  nameField.type("1");

  // THEN the 'Player Name' field should not change
  nameField.invoke("val").should("equal", "1234567890");
});
```

- 6.2. Run the test suite once more. All of the tests should pass.

```
[user@host scoreboard]$ npm run cy:run
...output omitted...
      Spec                                Tests  Passing  Failing  Pending  Skipped
┌ #  scoreboard.spec.ts      00:01      4        4        -        -        - ─┘
# All specs passed!      00:01      4        4        -        -        -
```

► **7.** Stop Cypress and the development server.

- 7.1. If you ran Cypress graphically with `cy:open`, then close all Cypress and web browser windows. Skip this step if you ran Cypress in another mode.
- 7.2. In the terminal where it is running, shut down the development server by pressing `CTRL + c`

► **8.** Commit all the changes to the exercise branch.

```
[user@host scoreboard]$ git add .
...output omitted...
[user@host scoreboard]$ git commit -m "finish exercise"
...output omitted...
```

This concludes the guided exercise.

► Lab

Implementing Unit, Integration, and Functional Testing for Applications

In this lab, you will improve the tests available in the `calculator-microservices` application. This is a microservices version of the Quarkus Calculator.

The structure of this application is:

- **Solver:** if the formula contains a sum or a multiplication, passes the operation to the appropriate service.
- **Adder:** solves the received sum, and defers further calculations to the solver service.
- **Multiplier:** solves the received multiplication, and defers further calculations to the solver service.

You will mock the `adder` and `multiplier` services, create unit and integration tests for the `solver` service, and functional tests for the whole application.

You will find the solution files for this exercise in the `solutions` branch of the `D0400-apps` repository, within the `calculator-microservices` folder.

Outcomes

You should be able to mock services and create unit tests, integration tests, and functional tests.

Before You Begin

If not done before, fork the <https://github.com/RedHatTraining/D0400-apps> repository into your own GitHub account, clone the fork locally, and move to the `main` branch.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Create a new branch named `testing-implementation`. Use the `calculator-microservices` folder containing the calculator application.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git remote -v
origin https://github.com/your_github_user/D0400-apps.git (fetch)
origin https://github.com/your_github_user/D0400-apps.git (push)
[user@host D0400-apps]$ cd calculator-microservices
[user@host calculator-microservices]$ git checkout main
Switched to branch 'main'
```

```
...output omitted...
[user@host calculator-microservices]$ git checkout -b testing-implementation
Switched to a new branch 'testing-implementation'
```

Instructions

1. Create unit tests for the `MultiplierResource` and mock the `SolverService` service in the `multiplier` service.
Create one unit test for a simple multiplication of two positive values, another one for one positive and one negative value, and one more for the case of an invalid value.
2. Create an integration test for the `AdderResource` and mock the `SolverService` services in the `adder` service. This test will verify the integration between components in this microservice, and mock the external services.
Create one integration test for a simple sum of two positive values, another one for one positive and one negative value, and one more for the case of an invalid value.
3. Create a functional test for the `Solver` service launching the `Adder` and `Multiplier` services.
Create one functional test for a simple sum of two positive values, another one for a simple multiplication, and one more for the case of an invalid value.
Before running any test, be sure to run in a new terminal the `./start-dependant.sh` script located in the `calculator-microservices` directory.

```
[user@host calculator-microservices]$ ./start-dependant.sh
```



Note

This script will not work on Windows systems unless you have WSL installed.

4. Commit all the changes to the exercise branch.

This concludes the lab.

► Solution

Implementing Unit, Integration, and Functional Testing for Applications

In this lab, you will improve the tests available in the `calculator-microservices` application. This is a microservices version of the Quarkus Calculator.

The structure of this application is:

- **Solver:** if the formula contains a sum or a multiplication, passes the operation to the appropriate service.
- **Adder:** solves the received sum, and defers further calculations to the solver service.
- **Multiplier:** solves the received multiplication, and defers further calculations to the solver service.

You will mock the `adder` and `multiplier` services, create unit and integration tests for the `solver` service, and functional tests for the whole application.

You will find the solution files for this exercise in the `solutions` branch of the `D0400-apps` repository, within the `calculator-microservices` folder.

Outcomes

You should be able to mock services and create unit tests, integration tests, and functional tests.

Before You Begin

If not done before, fork the <https://github.com/RedHatTraining/D0400-apps> repository into your own GitHub account, clone the fork locally, and move to the `main` branch.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Create a new branch named `testing-implementation`. Use the `calculator-microservices` folder containing the calculator application.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git remote -v
origin https://github.com/your_github_user/D0400-apps.git (fetch)
origin https://github.com/your_github_user/D0400-apps.git (push)
[user@host D0400-apps]$ cd calculator-microservices
[user@host calculator-microservices]$ git checkout main
Switched to branch 'main'
```

```
...output omitted...
[user@host calculator-microservices]$ git checkout -b testing-implementation
Switched to a new branch 'testing-implementation'
```

Instructions

1. Create unit tests for the `MultiplierResource` and mock the `SolverService` service in the `multiplier` service.

Create one unit test for a simple multiplication of two positive values, another one for one positive and one negative value, and one more for the case of an invalid value.

- 1.1. Enter the `multiplier` service directory.

```
[user@host calculator-microservices]$ cd multiplier
```

- 1.2. Open the `multiplier/src/main/java/com/redhat/training/MultiplierResource.java` file and observe the `multiply` method.
- 1.3. Open the unit test file `multiplier/src/test/java/com/redhat/training/MultiplierResourceTest.java`.
- 1.4. Create a unit test that verifies the `multiply` method returns 6 when given the parameters 2 and 3. Mock the `solverService` service to make it return the given parameter.

The test should be similar to:

```
@Test
public void simpleMultiplication() {
    // Given
    Mockito.when(solverService.solve("2")).thenReturn(Float.valueOf("2"));
    Mockito.when(solverService.solve("3")).thenReturn(Float.valueOf("3"));

    // When
    Float result = multiplierResource.multiply("2", "3");

    // Then
    assertEquals( 6.0f, result );
}
```

In case your IDE does not provide you with the correct imports, you can copy them from here:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Response;
import org.jboss.resteasy.client.exception.ResteasyWebApplicationException;
import org.junit.jupiter.api.function.Executable;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
```


Add the necessary imports, run the test and verify that it passes.

```
[user@host multiplier]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS

...output omitted...
```

- 1.5. Add a new test like the first one but passing a negative value.

The new test should be similar to:

```
@Test
public void negativeMultiply() {
    Mockito.when(solverService.solve("-2")).thenReturn(Float.valueOf("-2"));
    Mockito.when(solverService.solve("3")).thenReturn(Float.valueOf("3"));

    // When
    Float result = multiplierResource.multiply("-2", "3");

    // Then
    assertEquals( -6.0f, result );
}
```

Run the tests and verify that all tests pass.

```
[user@host multiplier]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS

...output omitted...
```

- 1.6. Add one more unit test that verifies that an exception is thrown when an invalid value is passed.

This test should be similar to:

```

@Test
public void wrongValue() {
    WebApplicationException cause = new WebApplicationException("Unknown error",
    Response.Status.BAD_REQUEST);
    Mockito.when(solverService.solve("a")).thenThrow( new
    ResteasyWebApplicationException(cause) );
    Mockito.when(solverService.solve("3")).thenReturn(Float.valueOf("3"));

    // When
    Executable multiplication = () -> multiplierResource.multiply("a", "3");

    // Then
    assertThrows( ResteasyWebApplicationException.class, multiplication );
}

```

Run the tests and verify that all tests pass.

```

[user@host multiplier]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS

...output omitted...

```

2. Create an integration test for the `AdderResource` and mock the `SolverService` services in the `adder` service. This test will verify the integration between components in this microservice, and mock the external services.

Create one integration test for a simple sum of two positive values, another one for one positive and one negative value, and one more for the case of an invalid value.

- 2.1. Enter the `adder` service directory.

```

[user@host multiplier]$ cd ../adder

```

- 2.2. Open the `adder/src/main/java/com/redhat/training/AdderResource.java` file and observe the `add` method.
- 2.3. Open the integration test file `adder/src/test/java/com/redhat/training/AdderResourceTest.java`.
- 2.4. Create an integration test that verifies the `add` method returns 5 when given the parameters 2 and 3. Mock the `solverService` service to make it return the given parameter.

The test should look be similar to:

```

@Test
public void simpleSum() {
    Mockito.when(solverService.solve("2")).thenReturn(Float.valueOf("2"));
    Mockito.when(solverService.solve("3")).thenReturn(Float.valueOf("3"));

    given()
        .when().get("3/2")
        .then()
            .statusCode(200)
            .body(is("5.0"));
}

```

If your IDE does not provide you with the correct imports, then you can copy them from here:

```

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Response;
import org.jboss.resteasy.client.exception.ResteasyWebApplicationException;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

```

Run the test and verify that it passes.

```

[user@host adder]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS

...output omitted...

```

2.5. Add a new test like the first one but passing a negative value.

The new test should be similar to:

```

@Test
public void negativeSum() {
    Mockito.when(solverService.solve("-2")).thenReturn(Float.valueOf("-2"));
    Mockito.when(solverService.solve("3")).thenReturn(Float.valueOf("3"));

    given()
        .when().get("3/-2")
        .then()

```

```

        .statusCode(200)
        .body(is("1.0"));
    }

```

Run the tests and verify that all tests pass.

```

[user@host adder]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
...output omitted...

```

- 2.6. Add one more integration test that verifies that an exception is thrown when an invalid value is passed.

This test should be similar to:

```

@Test
public void wrongValue() {
    WebApplicationException cause = new WebApplicationException("Unknown error",
        Response.Status.BAD_REQUEST);
    Mockito.when(solverService.solve("a")).thenThrow( new
        ResteasyWebApplicationException(cause) );
    Mockito.when(solverService.solve("3")).thenReturn(Float.valueOf("3"));

    given()
        .when().get("3/a")
        .then()
            .statusCode(Response.Status.BAD_REQUEST.getStatusCode());
}

```

Add the necessary imports, run the tests and verify that all tests pass.

```

[user@host adder]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
...output omitted...

```

3. Create a functional test for the `Solver` service launching the `Adder` and `Multiplier` services.

Create one functional test for a simple sum of two positive values, another one for a simple multiplication, and one more for the case of an invalid value.

Before running any test, be sure to run in a new terminal the `./start-dependant.sh` script located in the `calculator-microservices` directory.

```
[user@host calculator-microservices]$ ./start-dependant.sh
```



Note

This script will not work on Windows systems unless you have WSL installed.

- 3.1. Enter the `solver` service directory.

```
[user@host adder]$ cd ../solver
```

- 3.2. Open the `solver/src/main/java/com/redhat/training/SolverResource.java` file and observe the `solve` method.
- 3.3. Open the functional test file `solver/src/test/java/com/redhat/training/SolverResourceTest.java`.
- 3.4. Create a functional test that verifies the `add` method returns 5 when given the parameters 2 and 3. Mock the `solverService` service to make it return the given parameter.

The test should look be similar to:

```
@Test
public void simpleSum() {
    given()
    .when().get("3+2")
    .then()
    .statusCode(200)
    .body(is("5.0"));
}
```

If your IDE does not provide you with the correct imports, then you can copy them from here:

```
import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

import org.junit.jupiter.api.Test;
import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;
import javax.ws.rs.core.Response;
```

Add the necessary imports, run the test, and verify that it passes.

```
[user@host solver]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS

...output omitted...
```

- 3.5. Add a new test like the first one but for the multiplication solving.

The new test should be similar to:

```
@Test
public void simpleMultiplication() {
    given()
        .when().get("3*2")
        .then()
            .statusCode(200)
            .body(is("6.0"));
}
```

Run the tests and verify that all tests pass.

```
[user@host solver]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS

...output omitted...
```

- 3.6. Add one more functional test that verifies that an exception is thrown when an invalid value is passed.

This test should be similar to:

```

@Test
public void wrongValue() {
    given()
        .when().get("3*a")
        .then()
            .statusCode(Response.Status.BAD_REQUEST.getStatusCode());
}

```

Run the tests and verify that all tests pass.

```

[user@host solver]$ ./mvnw test
[INFO] Scanning for projects...

...output omitted...

[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS

...output omitted...

```

4. Commit all the changes to the exercise branch.

```

[user@host solver]$ cd ..
[user@host calculator-microservices]$ git commit -a -m "Add tests"

```

This concludes the lab.

Summary

In this chapter, you learned:

- Different types of tests validate software at different levels.
- Unit tests are the preferred way to validate specific code units.
- Integration tests validate that two or more components work together.
- Functional tests evaluate use cases treating the application as a black box.

Chapter 4

Building Applications with Test-driven Development

Goal

Implement and build application features with Test-driven Development.

Objectives

- Describe the process and implementation of test-driven development.
- Identify development best practices such as SOLID and Test-driven design, testing best practices and development design patterns.
- Evaluate code quality and test coverage by using linters and style checkers.

Sections

- Introducing Test-driven Development (and Guided Exercise)
- Developing with TDD and Other Best Practices (and Quiz)
- Analyzing Code Quality (and Guided Exercise)

Lab

Building Applications with Test-driven Development

Introducing Test-driven Development

Objectives

After completing this section, you should be able to describe the process and implementation of test-driven development.

Defining Test-driven Development

Test-driven Development (TDD) is a software development practice that uses short development cycles to drive the development of applications. With this approach, developers split the functionality specification derived from a use case, into smaller units of work and develop each of these units individually.

It is up to the developer in each particular case to decide what a unit consists. When testing REST APIs, a unit could be an endpoint; but when testing other software platforms a unit could be a method.

In his book *Extreme Programming Explained: Embrace Change*, Kent Beck explained that he rediscovered this software practice. He later further refined it in his book *Test-driven Development by Example*.

To start the development of one of these small units, the developer creates a test, which reflects a specific use case and executes it to verify that the new test fails. This is important because, in the case of a successful test, it means that the test is not testing the right behavior because a missing implementation should always lead to a failing test.

After verifying that the test fails, the developer creates enough implementation code to make the test evaluating this particular use case pass. In this step, the developer attempts to achieve a passing test as soon as possible without paying too much attention to the actual state of the code. This leads to messy code, which the developer will correct in the next step.

Once the code passes the test, the developer must refine the code and tests to remove duplicated or hard coded code, and refactor the code by splitting it into smaller functions or move it to a more appropriate place in the class hierarchy.

During this step the developer uses all the existing tests to check that the code still functions as expected.

Developers repeat these three steps until completing all the units derived from the user story. This process is commonly referred to as the TDD Cycle or as the Red, Green, and Refactor Cycle.

TDD Cycle

- Red: write a test that fails because the implementation is missing.
- Green: write the minimum code to make the test pass.
- Refactor: review the code and the tests to improve the solution.

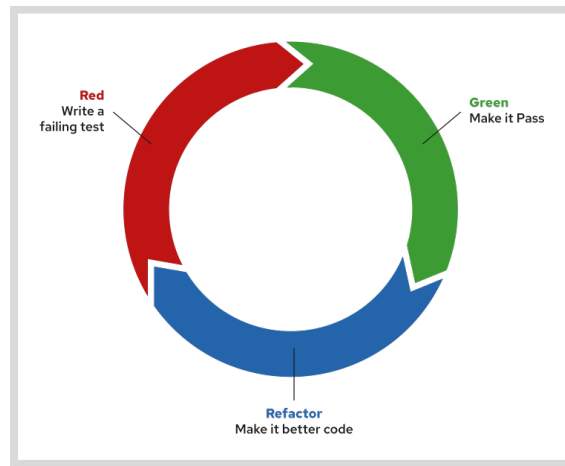


Figure 4.1: Red - Green - Refactor cycle

Describing the Benefits of TDD

TDD adoption brings advantages to the code that is written. It takes time to adapt to, because it switches the traditional focus to a test-first mentality, but once you get used to it you benefit from the improved workflow.

A non-comprehensive list of benefits of developing using TDD might be:

- The code reflects all the scenarios specified for the functionality.
- Reduces the number of bugs and the project cost in the long term.
- The existence of tests from the beginning helps to identify issues quickly.
- The final code is easier to refactor.

Applying Best Practices

As we covered in the testing chapter, several practices have been developed over years of encountering the same patterns while developing applications using TDD. These practices aim to offer the best developer experience while maximizing the benefits of using TDD.

Writing Well Structured Tests

A well structured test serves as self-documenting code for the unit of the use case implementation it is testing. The usual structure for a test in TDD derives from the Given-When-Then common structure for any test:

1. Set up the test scenario, which includes any test data or environment needed.
2. Execute the action to test and capture return values.
3. Validate the results asserting the return values.
4. Clean up the test scenario.

To start this structure the recommended practice is to start with the assertion, which tests the unit behavior. The second step is to call the code to test because you need the value to test in the assert just created.

The next step should be to add code to the module call to make the test pass. While creating this code be sure to mind two common software developer practices:

- Keep It Simple, Stupid (KISS): Do not complicate or over-engineer the code, write simple code to pass the test.
- You Aren't Gonna Need It (YAGNI): Do not add any code that is not necessary for the current test.

This practices will be further detailed in later lectures.

The final steps should be to set up the necessary code and data to support the test case, and clean up what you have just set up.



References

Test-driven development

https://en.wikipedia.org/wiki/Test-driven_development

TDD Best practices

<https://scand.com/company/blog/test-driven-development-best-practices/>

Kent Beck. Test-driven development by Example. 2nd Edition. 2003.

Kent Beck. Extreme Programming Explained: Embrace Change. 5th Edition. 2005

► Guided Exercise

Introducing Test-driven Development

In this exercise you will create a shipping calculator following the Test-driven development (TDD) process.

The shipping calculator returns a fixed cost for each one of the supported regions:

- NA: 100
- LATAM: 200
- EMEA: 300
- APAC: 400

Calling with an unsupported region raises an exception.

By using small TDD cycles, you will specify the preceding conditions as test cases, develop the functional code to make the tests pass, and refactor your code. You will start with a simple test covering only one of the regions. As you iterate over *Red*, *Green*, and *Refactor* cycles, you will add more regions until you develop the whole use case.

The source code of the shipping calculator is located in the GitHub repository at <https://github.com/RedHatTraining/D0400-apps> under the `shipping-calculator` directory.

You will find the solution files for this exercise in the `solutions` branch of the `D0400-apps` repository, within the `shipping-calculator` folder.

Outcomes

You should be able to follow the Red, Green, Refactor cycle to develop an application.

Before You Begin

To perform this exercise, ensure you have your `D0400-apps` fork cloned in your workspace, and the Java Development Kit (JDK) installed on your computer. You also need access to a command line terminal.

Use a text editor such as `VSCodium`, which supports syntax highlighting for editing source files.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the shipping calculator application.

- 1.1. From your workspace folder, navigate to the `D0400-apps/shipping-calculator` application folder and checkout the `main` branch of the `D0400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps/shipping-calculator
[user@host shipping-calculator]$ git checkout main
...output omitted...
```

- 1.2. Create a new branch to save any changes you make during this exercise.

```
[user@host shipping-calculator]$ git checkout -b tdd-intro
Switched to a new branch 'tdd-intro'
```

- ▶ 2. Create an initial test scenario that asserts the shipping cost for any region returns 0. The specification for the test is:

- Given a shipping calculator.
- When the shipping cost for a region is calculated.
- Then the shipping cost is 0.

This test serves as a starting point. It will be refactored or discarded in following TDD iterations.

- 2.1. Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and append the test implementation:

```
@Test
public void canCalculateTheCostForARegion() {
    ShippingCalculator calculator = new ShippingCalculator();

    assertEquals(0, calculator.costForRegion("A Region"));
}
```

The preceding test creates an instance of the shipping calculator. It checks the return of the `costForRegion` method for any region.

This is the initial test to establish arbitrary region values and their calculated shipping cost. The following TDD iterations will evolve with the source code and tests.

- 2.2. Run the tests and verify that the compilation fails because:

- There is no implementation of the shipping calculator.
- There is no implementation of the `costForRegion` method in the shipping calculator.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] COMPILATION ERROR :
...output omitted...
[ERROR] ...output omitted.../ShippingCalculatorTest.java:[12,9] cannot find symbol
symbol:   class ShippingCalculator
location: class com.redhat.shipping.ShippingCalculatorTest
```

```
[ERROR] ...output omitted.../ShippingCalculatorTest.java:[12,45] cannot find
symbol
  symbol:   class ShippingCalculator
  location: class com.redhat.shipping.ShippingCalculatorTest
...output omitted...
```

2.3. Write the minimum code to make the application compile.

Create the `src/main/java/com/redhat/shipping/ShippingCalculator.java` file with the following content:

```
package com.redhat.shipping;

public class ShippingCalculator {

    public int costForRegion(String name) {
        return 0;
    }
}
```

2.4. Run the tests and verify that the application compiles and the tests pass.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

2.5. Analyze the implementation code and the tests to find blocks of code to refactor.

At this point the code and the tests do not need a refactor.

► 3. Create a test for the scenario of calculating the shipping cost for the NA region. The specification for the test is:

- Given a shipping calculator.
- When the shipping cost for the NA region is calculated.
- Then the shipping cost is 100.

3.1. Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and append the test implementation:

```
@Test
public void onNARegionTheCostIs100() {
    assertEquals(100, (new ShippingCalculator()).costForRegion("NA"));
}
```

The preceding test checks that when the `costForRegion` method receives the NA string, then the returned value is 100.

3.2. Run the tests and verify that the application compiles but the tests fail.


```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] Failures:
[ERROR]   ShippingCalculatorTest.onNARegionTheCostIs100:18 expected: <100> but
    was: <0>
[INFO]
[ERROR] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

The test fails because the implementation for the NA case is missing in the source code.

3.3. Write the minimum code to make the test pass.

Open the `src/main/java/com/redhat/shipping/ShippingCalculator.java` file and update the `costForRegion` method:

```
public int costForRegion(String name) {
    if (name.equals("NA")) {
        return 100;
    }

    return 0;
}
```

The preceding code adds the implementation for the NA case.

3.4. Run the tests to verify that they pass.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

3.5. Analyze the implementation code and the tests to find blocks of code to refactor.

Potential code improvements include:

- Change the `onNARegionTheCostIs100` test to make it easier to understand.

Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and update the `onNARegionTheCostIs100` method.

```
@Test
public void onNARegionTheCostIs100() {
    // Given a shipping calculator
    ShippingCalculator calculator = new ShippingCalculator();

    // When NA is the country region
    int calculatedCost = calculator.costForRegion("NA");
```

```
// Then the shipping cost is 100
assertEquals(100, calculatedCost);
}
```

Note that the test now follows the Given-When-Then (GWT) structure. This structure makes tests easier to read, understand, and check specification coverage.

- 3.6. Run the tests to verify that the application continues to match the specifications after the refactor.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

- ▶ 4. Create a test for the scenario of calculating the shipping cost for the LATAM region. The specification for the test is:
 - Given a shipping calculator.
 - When the shipping cost for the LATAM region is calculated.
 - Then the shipping cost is 200.
- 4.1. Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and append the test implementation:

```
@Test
public void onLATAMRegionTheCostIs200() {
    // Given a shipping calculator
    ShippingCalculator calculator = new ShippingCalculator();

    // When LATAM is the country region
    int calculatedCost = calculator.costForRegion("LATAM");

    // Then the shipping cost is 200
    assertEquals(200, calculatedCost);
}
```

The preceding test follows the GWT structure and checks that, when the `costForRegion` method receives the LATAM string, then the returned value is 200.

- 4.2. Run the tests and verify that the application compiles but the tests fail.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] Failures:
[ERROR] ShippingCalculatorTest.onLATAMRegionTheCostIs200:37 expected: <200> but
was: <0>
[INFO]
[ERROR] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

The test fails because the implementation for the LATAM case is missing in the source code.

- 4.3. Write the minimum code to make the test pass.

Open the `src/main/java/com/redhat/shipping/ShippingCalculator.java` file and update the `costForRegion` method.

```
public int costForRegion(String name) {
    if (name.equals("NA")) {
        return 100;
    }

    if (name.equals("LATAM")) {
        return 200;
    }

    return 0;
}
```

The preceding code adds the implementation for the LATAM case.

- 4.4. Run the tests to verify that they pass.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

- 4.5. Analyze the implementation code and the tests to find blocks of code to refactor.

At this point the code and the tests do not need a refactor.

- 5. Create a test for the scenario of calculating the shipping cost for the EMEA region. The specification for the test is:

- Given a shipping calculator.
- When the shipping cost for the EMEA region is calculated.
- Then the shipping cost is 300.

- 5.1. Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and append the test implementation:

```
@Test
public void onEMEARegionTheCostIs300() {
    // Given a shipping calculator
    ShippingCalculator calculator = new ShippingCalculator();

    // When EMEA is the country region
    int calculatedCost = calculator.costForRegion("EMEA");
}
```

```
// Then the shipping cost is 300
assertEquals(300, calculatedCost);
}
```

The preceding test checks that, when the `costForRegion` method receives the EMEA string, then the returned value is 300.

5.2. Run the tests and verify that the application compiles but the tests fail.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] Failures:
[ERROR]   ShippingCalculatorTest.onEMEARegionTheCostIs300:49 expected: <300> but
was: <0>
[INFO]
[ERROR] Tests run: 4, Failures: 1, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

The test fails because the implementation for the EMEA case is missing in the source code.

5.3. Write the minimum code to make the test pass.

Open the `src/main/java/com/redhat/shipping/ShippingCalculator.java` file and update the `costForRegion` method.

```
public int costForRegion(String name) {
    if (name.equals("NA")) {
        return 100;
    }

    if (name.equals("LATAM")) {
        return 200;
    }

    if (name.equals("EMEA")) {
        return 300;
    }

    return 0;
}
```

The preceding code adds the implementation for the EMEA case.

5.4. Run the tests to verify that they pass.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

5.5. Analyze the implementation code and the tests to find blocks of code to refactor.

Because there is only one specification left that affects the implementation of the cost calculation, the refactor can be done in the next TDD iteration.

- ▶ 6. Create a test for the scenario of calculating the shipping cost for the APAC region. The specification for the test is:
 - Given a shipping calculator.
 - When the shipping cost for the APAC region is calculated.
 - Then the shipping cost is 400.
- 6.1. Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and append the test implementation:

```
@Test
public void onAPACRegionTheCostIs400() {
    // Given a shipping calculator
    ShippingCalculator calculator = new ShippingCalculator();

    // When APAC is the country region
    int calculatedCost = calculator.costForRegion("APAC");

    // Then the shipping cost is 400
    assertEquals(400, calculatedCost);
}
```

- 6.2. Run the tests and verify that the application compiles but the tests fail.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] Failures:
[ERROR]   ShippingCalculatorTest.onAPACRegionTheCostIs400:61 expected: <400> but
was: <0>
[INFO]
[ERROR] Tests run: 5, Failures: 1, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

The test fails because the implementation for the APAC case is missing in the source code.

- 6.3. Write the minimum code to make the test pass.

Open the `src/main/java/com/redhat/shipping/ShippingCalculator.java` file and update the `costForRegion` method.

```
public int costForRegion(String name) {
    if (name.equals("NA")) {
        return 100;
    }

    if (name.equals("LATAM")) {
        return 200;
    }
}
```

```

    if (name.equals("EMEA")) {
        return 300;
    }

    if (name.equals("APAC")) {
        return 400;
    }

    return 0;
}

```

The preceding code adds the implementation for the APAC case.

6.4. Run the tests to verify that they pass.

```

[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...

```

6.5. Analyze the implementation code and the tests to find blocks of code to refactor.

Some improvements to the code that can be added:

- Reduce the number of returning points in the `costForRegion` method.
- Reduce the number of conditional statements in the `costForRegion` method.
- Store together the country region with the fixed cost.

Open the `src/main/java/com/redhat/shipping/ShippingCalculator.java` file and update the persistence of the different regions and costs. Change the `costForRegion` method to store the regions by using a `Map`.

```

package com.redhat.shipping;

import java.util.HashMap;
import java.util.Map;

public class ShippingCalculator {

    public static final Map<String, Integer> REGIONS = new HashMap<>();

    public ShippingCalculator() {
        ShippingCalculator.REGIONS.put("NA", 100);
        ShippingCalculator.REGIONS.put("LATAM", 200);
        ShippingCalculator.REGIONS.put("EMEA", 300);
        ShippingCalculator.REGIONS.put("APAC", 400);
    }

    public int costForRegion(String name) {
        if (ShippingCalculator.REGIONS.containsKey(name)) {

```

```

        return ShippingCalculator.REGIONS.get(name);
    }

    return 0;
}
}

```

The preceding code stores the relation between the regions and the cost in a `Map`. The constructor of the shipping calculator assigns the cost to each one of the supported regions. This allows the region string to be used as a key to obtain the mapped value. This further allows the removal of unnecessary return statements and conditionals.

- 6.6. Run the tests to verify that the application continues to match the specifications after the refactor.

```

[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...

```

- ▶ 7. Create a test for the scenario of calculating the shipping cost for an unsupported country region. The specification for the test is:
 - Given a shipping calculator.
 - When the shipping cost for an unsupported region is calculated.
 - Then a `RegionNotFoundException` exception is raised.

This test refines the specifications of the initial `canCalculateTheCostForARegion` test.

- 7.1. Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and append the test implementation:

```

@Test
public void onNonSupportedRegionARegionNotFoundExceptionIsRaised() {
    ShippingCalculator calculator = new ShippingCalculator();

    assertThrows(
        RegionNotFoundException.class,
        () -> calculator.costForRegion("Unknown Region")
    );
}

```

- 7.2. Run the tests and verify that the compilation fails because:
 - There is no implementation of the `RegionNotFoundException` exception.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] ...output omitted.../ShippingCalculatorTest.java:[69,13] cannot find
symbol
  symbol:   class RegionNotFoundException
  location: class com.redhat.shipping.ShippingCalculatorTest
...output omitted...
```

- 7.3. Write the minimum code to make the application compile.
Create the `src/main/java/com/redhat/shipping/RegionNotFoundException.java` file and add the following code:

```
package com.redhat.shipping;

public class RegionNotFoundException extends Exception {
}
```

- 7.4. Run the tests and verify that the application compiles but the tests fail.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] Failures:
[ERROR] ...output omitted... RegionNotFoundException to be thrown, but nothing
was thrown.
[INFO]
[ERROR] Tests run: 6, Failures: 1, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

The test fails because the implementation of the `costForRegion` method does not throw an exception.

- 7.5. Write the minimum code to make the test pass.
Open the `src/main/java/com/redhat/shipping/ShippingCalculator.java` file and update the `costForRegion` implementation:

```
public int costForRegion(String name) throws RegionNotFoundException {
    if (ShippingCalculator.REGIONS.containsKey(name)) {
        return ShippingCalculator.REGIONS.get(name);
    }

    throw new RegionNotFoundException();
}
```

Instead of returning an arbitrary value, the preceding code raises an exception to better match the specification.

- 7.6. Run the tests to verify that the changes are breaking the compilation of the other tests.


```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] ...output omitted... Compilation failure:
[ERROR] ...output omitted... unreported exception
com.redhat.shipping.RegionNotFoundException; must be caught or declared to be
thrown
...output omitted...
```

The `costForRegion` method signature indicates that it raises an exception. To make the application compile, that exception needs to be caught or declared to be thrown in any block of code that uses the `costForRegion` method.

- 7.7. Update the tests to make the application compile.

Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and update the tests to allow the propagation of the exception:

```
...output omitted...
@Test
public void canCalculateTheCostForARegion() throws RegionNotFoundException {
...output omitted...

@Test
public void onNAREgionTheCostIs100() throws RegionNotFoundException {
...output omitted...

@Test
public void onLATAMRegionTheCostIs200() throws RegionNotFoundException {
...output omitted...

@Test
public void onEMEAREgionTheCostIs300() throws RegionNotFoundException {
...output omitted...

@Test
public void onAPACRegionTheCostIs400() throws RegionNotFoundException {
...output omitted...
```

- 7.8. Run the tests to verify that the application compiles but the tests still fail.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[ERROR] Errors:
[ERROR] ShippingCalculatorTest.canCalculateTheCostForARegion:13 » RegionNotFound
[INFO]
[ERROR] Tests run: 6, Failures: 0, Errors: 1, Skipped: 0
[INFO]
...output omitted...
```

The `canCalculateTheCostForARegion` test fails because the `costForRegion` implementation now throws an exception when an unsupported region is used and it does not return 0. This test can be safely removed because the other tests cover the rest of the shipping calculator requirements.

- 7.9. Open the `src/test/java/com/redhat/shipping/ShippingCalculatorTest.java` file and remove the `canCalculateTheCostForARegion` test.
- 7.10. Run the tests to verify that the application compiles and all tests pass.

```
[user@host shipping-calculator]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

- ▶ 8. Commit all the changes to the exercise branch.

```
[user@host shipping-calculator]$ git add .
...output omitted...
[user@host shipping-calculator]$ git commit -m "finish exercise"
...output omitted...
```

This concludes the guided exercise.

Developing with TDD and Other Best Practices

Objectives

After completing this section, you should be able to identify development best practices such as SOLID and Test-driven design, testing best practices and development design patterns.

Using Best Practices While Developing

For a long time developers and engineers have been facing recurring challenges during all phases of software development. To overcome those challenges, developers have defined several *development principles* as a set of golden rules to create better software. The following non-comprehensive list shows the most common development principles.

DRY (Don't repeat yourself)

Also known as the **Abstraction principle**, the DRY principle states that *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*.

This principle means that if two pieces of code share a common objective or pattern then the developer must refactor them to extract a parameterized abstraction of the pattern and then replace the original pieces to use the abstraction.

KISS (Keep It Simple Stupid)

Occam's razor of software development states that simplicity should be a design goal and developers must avoid unnecessary complexity.

YAGNI (You aren't gonna need it)

Originated in the **extreme programming** methodology, YAGNI advocates for avoiding any development that is not deemed necessary.

IoC (Inversion Of Control)

This principle promotes developing small business-specific building blocks while delegating the application workflow to a higher-purpose framework. This framework provides technical foundations such as application entry points or dependency life cycle management and injection.

IoC is frequently applied using *dependency injection*. With dependency injection, a client class, or function, is not responsible for creating instances of its dependencies. Instead, a higher-purpose IoC framework resolves, creates, and injects the dependencies into the client, decoupling this client from its dependencies. This also allows your code to depend on abstractions, or interfaces, rather than specific implementations.

Implementations of these IoC frameworks exist in multiple languages. For example, in Java, you can use the *Contexts and Dependency Injection* (CDI) framework.

SOLID

Not a single principle, but a set of principles covering different aspects of software development:

- **S**: Single responsibility principle (SRP)

Every unit of code must encapsulate a one and just one functionality. In the words of Robert C. Martin, *A class should have only one reason to change*. This emphasizes the fact that functionality changes must impact a single class per functionality.

- **O**: Open–closed principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. That is, the unit must be reasonably extendable for adapting its behavior to different implementation details, and must also have an immutable behavior, which is predictable and usable by other units.

- **L**: Liskov substitution principle (LSP)

Also known as principle of substitutability or strong behavioural subtyping. This principle defends that if a type `T` presents some desirable and provable properties then any type `S` subtype of `T` must also present those same properties. In other words, developers must be able to replace instances of `T` by instances of any type `S` retaining the desirable properties.

For example, if you have a method that accepts a `Vehicle` type, then you should be able to call this method with subtypes of `Vehicle`, such as `Car` and `Motorcycle`, without breaking your method.

- **I**: Interface segregation principle (ISP)

This principle is a SRP variant for interfaces. It states that interface clients must depend only on interface methods related to the functionality it is actively using, and not depend on other functionalities. This leads to requiring the interface to provide a single functionality, thus splitting interfaces with multiple functions.

For example, the following code breaks this principle because sharks cannot run.

```
interface Animal {
    void run();
    void swim();
}

class Shark implements Animal {
    void run() {
        throw new Exception("I cannot run");
    }
    void swim() {
        // implementation
    }
}
```

You can fix the example by splitting the interface as follows:

```
interface Runner {
    void run();
}

interface Swimmer {
    void swim();
}

class Shark implements Swimmer {
    void swim() {
        // implementation
    }
}
```

```

}

class Dog implements Runner, Swimmer {
    void run() {
        // implementation
    }
    void swim() {
        // implementation
    }
}

```

- **D:** Dependency inversion principle (DIP)

Robert C. Martin defined this principle with two declarations: - High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces). - Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

Other best practices not categorized as principles are:

Code Smells

Code smells refer to subjective indicators, which despite the code being technically correct might not honor some principles and unexpected issues might arise afterward. Examples of code smells are: duplicated code sections, methods with a large number of lines or input parameters, or complex `if` or `switch` statements.

Many tools can detect code smells directly on the code base, facilitating the detection to the developer: PMD or Checkstyle for Java, ESLint for NodeJS, Pylint for Python, or SonarQube for cross-language smell detection.

Decoupling

Many patterns previously presented targets to eliminate coupling between different code units. In general, this same idea of decoupling units applies at both code and architecture levels. For example, architectural patterns such as the MVC pattern decouple the user interface from the business logic and from the data.

Decoupling code units is especially important for testing patterns. If dependencies are correctly declared and decoupled, then unit and integration tests can replace those dependencies with test doubles and create more meaningful tests.

Testing with Best Practices

Software design patterns and principles are best practices for creating better software. Some best practices focus on the testing phase, looking for creating better tests. Some of those testing best practices are:

Quick, short, and independent unit tests

As already covered in *Describing Unit Tests* section, tests must be small, fast, isolated, stateless, and idempotent. Tests should fail fast and clearly, giving maintainers all the information they might need.

Short TDD iterations

Keep the TDD cycle of Red-Green-Refactor as small as possible. Small changes are easier to handle and to solve than big changes.

Tests as specifications

Keep a tight relation between your tests and your application requirements. Every requirement of your application must have at least one test; and every test must cover a single requirement.

When tests use the appropriate expression language, such as the **Given-When-Then**, and the right level of abstraction, tests become an expression of the application specifications.

Run tests after a change (local/CI)

Design tests with two execution points in mind: First, the development phase, or the TDD cycle; Developers should run the tests locally in the development workstations, so tests should be easy to integrate in the development workflow. Second, the CI environment; Tests must run in an automated fashion, cover the whole spectrum of tests, and produce automated alerts on failures.

Test-Driven-Design

Design your application with tests in mind. Code units must clearly declare and expose their dependencies, so tests can replace them as needed. Dependency injection frameworks are useful for that purpose.

Code must avoid private units when white box testing applies.

Design code units as stateless units so tests can declare the state under test.

Test like you code

The same best practices apply for application development and for test development. Test developers must honor principles such as **DRY**, **KISS**, **YAGNI**, and **SOLID**.

Decoupling

In general, cleaner tests lead to simpler, decoupled, and more maintainable production code. If you follow TDD, then at some point you might find that tests *drive* the architecture of your application towards more decoupled, maintainable code components.

In the same way, if your production code follows best practices and development principles, then tests will be easier to write and maintain. Best practices applied to both test and production code result in overall benefits to your development process.

In particular, principles such as **Inversion of Control** and **SOLID** help you to create cleaner, more independent, and more maintainable unit tests.

Identifying Development Design Patterns

Design patterns help developers solve common problems in software engineering.

These patterns are well-known solutions to common problems or feature requirements that are inline with development and design principles. Many of these patterns originate from object-oriented programming but also apply to other programming paradigms, such as functional programming. Most design patterns help implement or are canonical examples of software development principles, such as the **DRY** or **SOLID** principles.

The use of design patterns works well in combination with TDD, specially in the refactor step. Refactoring your code is an opportunity to apply design patterns to improve both the production and test code. For example, the **Builder** pattern is common in unit tests. It helps you create complex objects and abstracts the initialization logic. When used in unit tests, it allows you to simplify scenario setups, making tests cleaner.

There are many design patterns, and covering each of them in detail is outside of the scope of this course. However, you are encouraged to explore them and choose the patterns that best suit your needs.



References

Software design pattern

https://en.wikipedia.org/wiki/Software_design_pattern

S.O.L.I.D. principles

<https://en.wikipedia.org/wiki/SOLID>

Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. pp. 127–131. ISBN 978-0135974445.

Code Smells

<https://refactoring.guru/refactoring/smells>

Erich Gamma, Ralph Johnson, John Vlissides, Richard Helm. (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. ISBN 978-0201633612.

► Quiz

Developing with TDD and Other Best Practices

In this quiz, consider you are developing a home automation application to control a lighting system based on environment conditions. This application controls the lighting system based on the amount of daylight. Assume the application controls a fictional *Acme Lighting System*.

The daylight intensity is expressed as a factor between zero and one. A value of zero means no light, whereas a value of one indicates the maximum factor of daylight. Values equal to or below 0.20 indicate low daylight conditions.

Choose the correct answers to the following questions:

- 1. **You have to develop the use case that switches on the lights when daylight is equal to or below 0.20. Assuming you follow Test-driven Development (TDD) and best practices, which two of the following are a good first step? (Choose two.)**
- a. Write a function to switch on the lights when daylight equals 0.20.
 - b. Write a test to specify that the application switches on the lights when daylight equals 0.20.
 - c. Write a function to switch on the lights when daylight is below 0.20.
 - d. Write a test to specify that the application switches on the lights when daylight is below 0.20.
 - e. Write a test to specify how the application handles daylight values equal to, below, and above 0.20.

- 2. Assume you have just finished writing the test and production code to switch on the light under low daylight conditions. You now have two passing tests, covering this use case, as the following pseudo code shows. How would you refactor this code?

```
test_switch_on_lights_under_daylight_threshold():
    """Switch on the lights when daylight == 0.2"""
    light = AcmeLightSystem()
    home = HomeAutomation(light)

    # GIVEN daylight is 0.20
    daylight = 0.20

    # WHEN home automation processes the daylight
    home.process(daylight)

    # THEN light should be ON
    assert light.isOn()

test_switch_on_lights_under_low_daylight():
    """Switch on the lights when daylight < 0.2"""
    light = AcmeLightSystem()
    home = HomeAutomation(lights)

    # GIVEN daylight is below 0.20
    daylight = 0.19

    # WHEN home automation processes the daylight
    home.process(daylight)

    # THEN light should be ON
    assert light.isOn()
```

- Start a new TDD cycle to develop more functionality. No refactor needed.
- Join the two tests to unify the responsibilities of each of them into a single test that covers both scenarios.
- Apply the Don't Repeat Yourself (DRY) principle to avoid repeating the same initialization of the `light` and `home` variables. Extract the initialization of these variables to a separate function.
- Unify the responsibilities of the `AcmeLightSystem` and `HomeAutomation` classes into a single class. In this way, tests do not have to initialize two different objects.

- 3. Your production code includes the `HomeAutomation` class and the `AcmeLightSystem` class. The `HomeAutomation` class implements the rules to switch lights on and off, based on the amount of daylight. The `AcmeLightSystem` class acts as the adapter to your home lighting system. As the following pseudo code shows, the `HomeAutomation` class depends on the `AcmeLightSystem` class to control the lights. Which two of the following statements about the `HomeAutomation` class are correct? (Choose two.)

```
class AcmeLightSystem:
    switchOff():
        # Implementation ommitted

    switchOn():
        # Implementation ommitted

class HomeAutomation {
    lightSystem: AcmeLightSystem

    process(daylight):
        # Control 'lightSystem', based on 'daylight'
        # Implementation ommitted
```

- The `HomeAutomation` class is decoupled from the lighting system and easily adapts to other lighting system vendors.
- The `HomeAutomation` class is coupled to a particular lighting system and does not easily adapt to other lighting system vendors.
- The code structure facilitates the task of testing the `HomeAutomation` class in isolation.
- The `AcmeLightSystem` dependency makes unit tests potentially slower and more difficult to setup.
- Unit tests can easily replace the `AcmeLightSystem` dependency to validate the connection with alternative lighting systems.

- 4. Considering the example from the previous question, which two of the following are principles or best practices to apply to the `HomeAutomation` class? (Choose two.)
- a. Run unit tests less often, so that you do not overload your home lighting system with test requests.
 - b. Install a secondary lighting system in your home, dedicated to unit testing. In this way you do not alter the performance of the primary lighting system.
 - c. Use the `Dependency inversion` principle. Make the `HomeAutomation` class depend on an abstraction, the `LightSystem` interface, instead of the concrete `AcmeLightSystem` class.
 - d. When unit testing the `HomeAutomation` class, use a test double to decouple the test from a specific lighting system.
 - e. Apply the `Don't repeat yourself` (DRY) principle. Move the implementation of the `AcmeLightSystem` class into the `HomeAutomation` class to prevent repetition.

► Solution

Developing with TDD and Other Best Practices

In this quiz, consider you are developing a home automation application to control a lighting system based on environment conditions. This application controls the lighting system based on the amount of daylight. Assume the application controls a fictional *Acme Lighting System*.

The daylight intensity is expressed as a factor between zero and one. A value of zero means no light, whereas a value of one indicates the maximum factor of daylight. Values equal to or below 0.20 indicate low daylight conditions.

Choose the correct answers to the following questions:

- 1. **You have to develop the use case that switches on the lights when daylight is equal to or below 0.20. Assuming you follow Test-driven Development (TDD) and best practices, which two of the following are a good first step? (Choose two.)**
- a. Write a function to switch on the lights when daylight equals 0.20.
 - b. Write a test to specify that the application switches on the lights when daylight equals 0.20.
 - c. Write a function to switch on the lights when daylight is below 0.20.
 - d. Write a test to specify that the application switches on the lights when daylight is below 0.20.
 - e. Write a test to specify how the application handles daylight values equal to, below, and above 0.20.

- 2. Assume you have just finished writing the test and production code to switch on the light under low daylight conditions. You now have two passing tests, covering this use case, as the following pseudo code shows. How would you refactor this code?

```
test_switch_on_lights_under_daylight_threshold():
    """Switch on the lights when daylight == 0.2"""
    light = AcmeLightSystem()
    home = HomeAutomation(light)

    # GIVEN daylight is 0.20
    daylight = 0.20

    # WHEN home automation processes the daylight
    home.process(daylight)

    # THEN light should be ON
    assert light.isOn()

test_switch_on_lights_under_low_daylight():
    """Switch on the lights when daylight < 0.2"""
    light = AcmeLightSystem()
    home = HomeAutomation(lights)

    # GIVEN daylight is below 0.20
    daylight = 0.19

    # WHEN home automation processes the daylight
    home.process(daylight)

    # THEN light should be ON
    assert light.isOn()
```

- Start a new TDD cycle to develop more functionality. No refactor needed.
- Join the two tests to unify the responsibilities of each of them into a single test that covers both scenarios.
- Apply the Don't Repeat Yourself (DRY) principle to avoid repeating the same initialization of the `light` and `home` variables. Extract the initialization of these variables to a separate function.
- Unify the responsibilities of the `AcmeLightSystem` and `HomeAutomation` classes into a single class. In this way, tests do not have to initialize two different objects.

- 3. Your production code includes the `HomeAutomation` class and the `AcmeLightSystem` class. The `HomeAutomation` class implements the rules to switch lights on and off, based on the amount of daylight. The `AcmeLightSystem` class acts as the adapter to your home lighting system. As the following pseudo code shows, the `HomeAutomation` class depends on the `AcmeLightSystem` class to control the lights. Which two of the following statements about the `HomeAutomation` class are correct? (Choose two.)

```
class AcmeLightSystem:
    switchOff():
        # Implementation ommitted

    switchOn():
        # Implementation ommitted

class HomeAutomation {
    lightSystem: AcmeLightSystem

    process(daylight):
        # Control 'lightSystem', based on 'daylight'
        # Implementation ommitted
```

- The `HomeAutomation` class is decoupled from the lighting system and easily adapts to other lighting system vendors.
- The `HomeAutomation` class is coupled to a particular lighting system and does not easily adapt to other lighting system vendors.
- The code structure facilitates the task of testing the `HomeAutomation` class in isolation.
- The `AcmeLightSystem` dependency makes unit tests potentially slower and more difficult to setup.
- Unit tests can easily replace the `AcmeLightSystem` dependency to validate the connection with alternative lighting systems.

- 4. Considering the example from the previous question, which two of the following are principles or best practices to apply to the `HomeAutomation` class? (Choose two.)
- a. Run unit tests less often, so that you do not overload your home lighting system with test requests.
 - b. Install a secondary lighting system in your home, dedicated to unit testing. In this way you do not alter the performance of the primary lighting system.
 - c. Use the **Dependency inversion** principle. Make the `HomeAutomation` class depend on an abstraction, the `LightSystem` interface, instead of the concrete `AcmeLightSystem` class.
 - d. When unit testing the `HomeAutomation` class, use a test double to decouple the test from a specific lighting system.
 - e. Apply the **Don't repeat yourself** (DRY) principle. Move the implementation of the `AcmeLightSystem` class into the `HomeAutomation` class to prevent repetition.

Analyzing Code Quality

Objectives

After completing this section, you should be able to evaluate code quality and test coverage by using linters and style checkers.

Defining Code Quality

Code quality refers to a measure of how software meets nonfunctional requirements such as reliability, maintainability, and compliance with given design patterns. Developers use several *Static Testing* techniques to assess the quality of the code depending on which aspect of the code they need to verify. This lecture reviews the most basic and important ones: Code Conventions, Code Coverage, and Cyclomatic Complexity.

Numerous tools exist to assert code quality on different languages, but in this lesson you are going to learn about three, which target the Java language: CheckStyle, *Programming Mistake Detector* (PMD), and Jacoco.

Another tool worth mentioning is SonarQube, a cross-language code style and quality checker. SonarQube is out of the scope of this course, but you are encouraged to explore its features and possibilities as a code quality analyzer.

There might be some overlap between these tools because sometimes categorization for the rules enforced by each one differ and, as usual, there is no silver bullet in static code testing: it is highly recommended to find the right combination of them for each project.

Benefits of Code Quality

- Increases the readability of the code.
- Improves the reliability and robustness of the applications.
- Reduces technical debt.
- Improves the maintenance of the code.

Integrating Code Conventions

Code conventions are a set of rules, guidelines, and best practices to write programs in a specific language. Each language has some well-known code conventions, which serve as a baseline from where each team can establish their own conventions.

Code conventions can refer to simple rules about the code style such as code structure conventions or best practices. Some examples of code convention rules are:

Code style:

- Indentation
- Use of white spaces vs tabs
- Line length

Code structure

- Naming conventions
- Comments format
- Placement of declaration or statements

Best Practices

- Programming practices
- Architecture adherence
- Common pitfalls

You can enforce code conventions while coding, by using your preferred IDE, or during CI. Enforcing code conventions while coding makes reviewing code merges easier while enforcing them on the CI pipeline makes it easier to set up new developers because you avoid some configuration. Deciding which one to choose must be a team decision depending on its requirements.

Benefits of using Code Conventions

- Improves the readability of the code.
- Reduces the maintenance cost.
- Reduces signal-to-noise ratio during merges and reviews.

CheckStyle

To check the proper use of code conventions in your source code, you can use the CheckStyle Maven plug-in. To add this plug-in, you must add its configuration in the `plugins` section of your `pom.xml` file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId> ❶
  <version>3.1.1</version>
  <configuration>
    <consoleOutput>true</consoleOutput>
    <failsOnError>false</failsOnError>
    <linkXRef>false</linkXRef>
  </configuration>
  <executions>
    <execution>
      <id>validate</id>
      <phase>validate</phase>
      <goals>
        <goal>check</goal> ❷
      </goals>
    </execution>
  </executions>
</plugin>
```

- ❶** Adds the CheckStyle Maven plug-in

- ② Binds the check goal to the `validate` phase to run it before compilation.

**Note**

You can add the option `<failOnViolation>false</failOnViolation>` to the plug-in settings to not stop the build life-cycle on a violation.

Once you have configured the CheckStyle plug-in, run it with the `checkstyle:check` goal:

```
[user@host ~]$ ./mvnw clean checkstyle:check
```

By default the Maven CheckStyle plug-in uses the Sun Code Conventions but you can use any other or create one with your specific conventions. In the output of the command, you will find all the convention infractions in the code, you must inspect them individually to fix them, because each one is different.

Rule suppression

Not all code convention infractions must be addressed because some might not apply to each specific case. You must study each one of them and verify that it is actually an occurrence and not a false positive.

In case of common false positives or rules not applying your code style, you can disable rules for your project by using the `checkstyle-suppressions.xml` file in the root of the project. The contents of this file determine which rules to suppress and in which files. For example:

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
    "-//Checkstyle//DTD SuppressionFilter Configuration 1.2//EN"
    "https://checkstyle.org/dtds/suppressions_1_2.dtd">
<suppressions>
  <suppress
    checks="FinalClass" ①
    files=".*Resource.java" ②
  />
</suppressions>
```

- ① Suppress the "classes must be final" rule.
- ② Suppress the rule for all files ending in `Resource.java`

Verifying Code Coverage

Code coverage is a measure of the percentage of the source code executed when a test suite runs. A higher percentage means a lower chance of failure, but not that the code covered by the test suite is free of bugs.

Depending on how you measure the code coverage there are several criteria that can be used to identify the percentage of coverage.

Coverage Criteria

- Function coverage: Number of functions called over the total.

- Statement coverage: Number of statements called against the total.
- Branch coverage: Number of branches called versus the total.
- Condition coverage: Number of boolean conditions verified over total.
- Line coverage: Number of lines checked against the total.

Jacoco

In Java the most common tool to perform code coverage on source code is Jacoco. Jacoco has plug-ins for several IDEs and also for Maven.

To configure it in on Maven you have to add the plug-in to the `pom.xml` configuration file:

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.5</version>
  <executions>
    <execution>
      <id>default-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>default-report</id>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

After that, just execute the Maven `verify` phase to obtain the Jacoco report in the output directory.

```
[user@host ~]$ ./mvnw clean verify
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

This command outputs the report in the default folder `target/site/jacoco` in HTML format, so you can open the `index.html` file with any browser.

For each package in the analyzed source code you get a report, which itself contains a report on all the classes in that package. Then, for each class you get a report for all methods in that class. In each level you get the Missed Instructions and Missed Branches, and the percentage over the total for each metric.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
SampleClass		70 %		50 %
AdvancedSampleClass		97 %		95 %
Total	11 of 118	90 %	2 of 24	91 %

Figure 4.2: Sample code coverage

You have to investigate each particular case to identify why that particular area of code is not covered. Keep in mind that while you must aim for the highest percentage possible, 100% coverage is almost impossible to achieve.

Reducing Cyclomatic Complexity

The cyclomatic complexity is a metric for code quality, which determines the complexity of a fragment of code. A highly complicated code is more difficult to understand, and thus, more difficult to maintain and find issues.

Benefits of reducing the cyclomatic complexity

- Reduces the coupling in the code.
- The code is easier to understand.
- The code is easier to test.
- Reduces the maintenance cost.

To measure the cyclomatic complexity of any piece of code, you must count each possible path the execution can take. For example, a code with no execution branch or boolean check has a cyclomatic complexity of 1 because the execution path has only one route to follow. On the contrary, a code with one conditional branch such as the following has a cyclomatic complexity of 2 because it can either print the message or not:

```
public void conditionalGreet( boolean greet ) {
    if( greet ) {
        system.out.println( "Hi!" );
    }
}
```

Simple loops such as `foreach` or one condition `while` statements increase the cyclomatic complexity in 1, `while` loops with more than one condition use the same rules as conditional branching, plus one for the loop itself.

For more complex methods, you should calculate the execution flow graph of the method, and use the edges and nodes graph to calculate the complexity. When applied to a single method, the cyclomatic complexity M can be calculated as $M = \text{Edges} - \text{Nodes} + 2$. For example, assume that you want to analyze a conditional branch with two conditions.

```
public void conditionalGreet( boolean active, boolean greet ) {
    if( active && greet ) {
        system.out.println( "Hi!" );
    }
}
```

The following diagram represents the execution flow graph of the preceding method.

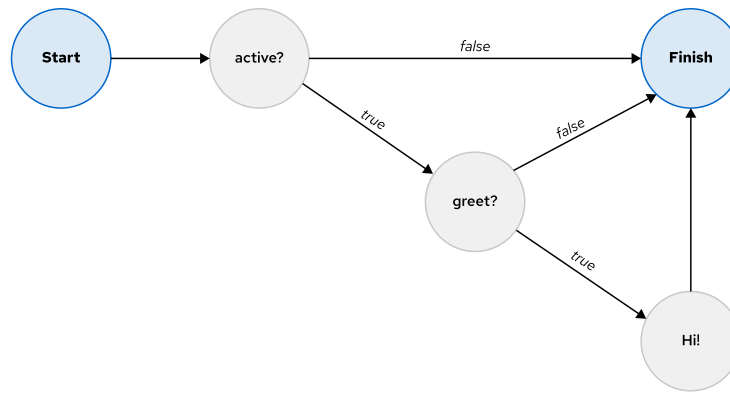


Figure 4.3: Execution flow graph

In this case, there are six edges and five nodes, therefore, $6 - 5 + 2$ results in a cyclomatic complexity of 3.

For methods, a commonly accepted cyclomatic complexity is 10 or less, but you can raise it to 15 for difficult code bases.

PMD

PMD is a tool you can use to check the cyclomatic complexity of your source code. As with other tools, you can use it embedded in your IDE or as part of your CI pipeline to avoid pushing code to the repository with unchecked issues.

To add PMD as part of your Maven build, you need to add its plug-in configuration to the plug-ins section in the `pom.xml` file:

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId> ❶
  <version>3.13.0</version>
  <configuration>
    <minimumTokens>10</minimumTokens>
    <printFailingErrors>true</printFailingErrors>
    <linkXRef>false</linkXRef>
    <rulesets>
      <ruleset>${pom.basedir}/pmd-rulesets.xml</ruleset> ❷
    </rulesets>
  </configuration>
</plugin>

```

- ❶ The Maven PMD plug-in
- ❷ Location of the PMD rule set file.

The PMD rule set file contains all the rules you want to check on the source code when executing the `pmd:pmd` phase. To check the cyclomatic complexity, you have to add the following rule to the `pmd-ruleset.xml` file:

```
<?xml version="1.0"?>
<ruleset name="PMD-Rules">
  <description>PMD rules</description>
  <rule ref="category/java/design.xml/CyclomaticComplexity" />
</ruleset>
```

The output of the `./mvnw pmd:pmd` command is a report like the Jacoco one in the `target/site/` folder. You can open the `pmd.html` file in your favorite browser and verify if any method or class has a cyclomatic complexity over the threshold to refactor them to simplify the code.

Removing Code Duplicates

Consider two pieces of code duplicated when they serve the same purpose and share the same implementation. The presence of duplicated code poses a problem because when an issue arises during the development or maintenance of the software in one of the instances, it is probable that the same issue is present in the duplicate code. It is common to address the issue in one of the versions of the code and forget about applying the same fix to its duplicates, thus leaving an error unfixed, which you believe is already fixed.

To avoid this, you must refactor any code duplication as soon as you detect it.

To detect these code duplicates, you can use PMD, which has a specialized check to find them. Once you have the PMD Maven plug-in configured, run the `pmd:cpd-check` goal to generate a report in the `target/site/cpd.html` file. Use your preferred browser to open the report and find any duplicated code in your application.

The easiest way to solve this situation is to refactor the duplicated method into a new method or class and call it from where you have removed it.

Performing Code Reviews

Automated code verification as you have seen in this lecture work well for catching most common issues, but it is far from perfect. This tool detects well-known code smells or bad practices, but subtle variations can pass under the radar and go unnoticed.

For this reason, everyone's work needs a second pair of eyes to perform peer validation. This reviewing process is a code review, also called a peer review.

Peers perform a code review by manually reviewing a change set, a commit, or a pull request in Git, for issues that fall into several categories:

- Possible defects: reviewers find issues that the author might have overlooked.
- Improve code quality: The resulting code is better thanks to the combined expertise.
- Knowledge inter-exchange: both author and reviewers learn from the inter-exchange that occurred during the review.
- Alternative solutions: better approaches lead to fewer bugs or better performance.

Author and reviewers conduct the process in iterative form, where they discuss and propose changes until they come to an agreement and merge the change.

In Agile teams, code reviews improve the sense of collective ownership of features and business domains.



References

Software quality

https://en.wikipedia.org/wiki/Software_quality

Coding conventions

https://en.wikipedia.org/wiki/Coding_conventions

Code coverage

https://en.wikipedia.org/wiki/Code_coverage

Cyclomatic complexity

https://en.wikipedia.org/wiki/Cyclomatic_complexity

Code review

https://en.wikipedia.org/wiki/Code_review

CheckStyle

<https://checkstyle.org/>

Jacoco

<https://www.jacoco.org/jacoco/>

PMD

<https://pmd.github.io/>

SonarLint

<https://www.sonarlint.org/>

► Guided Exercise

Analyzing Code Quality

In this exercise you will review the code style of a calculator, detect code duplication's, discover fragments of code with a high cyclomatic complexity, and measure the test coverage of the application.

You will find the solution files for this exercise in the `solutions` branch of the `D0400-apps` repository, within the `simple-calculator` folder.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (```) instead of the backslash character (`\`) to introduce line breaks in these commands.

Outcomes

You should be able to standardize the style of your code, find possible duplication's, detect code smells, and measure the coverage of your tests.

Before You Begin

To perform this exercise, ensure you have your `D0400-apps` fork cloned in your workspace, Git, and the Java Development Kit (JDK) installed on your computer. You also need access to a command line terminal and a web browser.

Use a text editor such as VSCode, which supports syntax highlighting for editing source files.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Get the source code of the sample application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps` application folder and checkout the `main` branch to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
...output omitted...
```

- 1.2. Copy the contents of `~/D0400/D0400-apps/simple-calculator` to the `~/D0400/quality-checks` directory. Navigate to the new directory.


```
[user@host D0400-apps]$ cp -Rv ~/D0400/D0400-apps/simple-calculator/ \
~/D0400/quality-checks
...output omitted...
[user@host D0400-apps]$ cd ~/D0400/quality-checks
[user@host quality-checks]$
```

**Important**

Windows users must replace the preceding `cp` command in PowerShell as follows:

```
PS C:\Users\user\D0400\D0400-apps> Copy-Item -Path simple-calculator `
>> -Destination ~\D0400\quality-checks -Recurse
```

► 2. Analyze and standardize the code style of the calculator.

- 2.1. Open the `pom.xml` file and add the Checkstyle plugin to the `build.plugins` section.

```
...output omitted...
<build>
  <plugins>
    ...output omitted...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>3.1.1</version>
      <configuration>
        <consoleOutput>>true</consoleOutput>
        <failsOnError>>false</failsOnError>
        <linkXRef>>false</linkXRef>
      </configuration>
      <executions>
        <execution>
          <id>validate</id>
          <phase>validate</phase>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    ...output omitted...
  </plugins>
</build>
...output omitted...
```

- 2.2. Run the Maven `checkstyle:check` goal to check if the code follows the Sun Code Conventions.

```
[user@host quality-checks]$ ./mvnw clean checkstyle:check
...output omitted...
[ERROR] Failed to execute ...output omitted... You have 48 Checkstyle violations.
...output omitted...
```

► 3. The execution of Checkstyle flags the following issues in the source code to address:

- There are some missing JavaDocs.
- The `BasicCalculator.java` file does not end with a newline.
- The imports should avoid the use of `*`.
- In the `BasicCalculator` class some methods can be declared as `final` if they are not going to be extended.
- The parameters of some methods in the `BasicCalculator` class can be declared as `final`.
- The `if` construct must use curly braces.
- There are some missing whitespaces.
- The `Random` method is not following the naming standard.

You can locate the majority of the issues in the `src/main/java/com/redhat/simple/calculator/BasicCalculator.java` file.

- 3.1. Create a file named `checkstyle-suppressions.xml`, in the root of the project, to store all the suppression filters. Add a suppression filter for all the JavaDoc comments. The file contents should be similar to:

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
    "-//Checkstyle//DTD SuppressionFilter Configuration 1.2//EN"
    "https://checkstyle.org/dtds/suppressions_1_2.dtd">
<suppressions>
  <suppress checks="Javadoc" files="."/>
</suppressions>
```

- 3.2. Open the `pom.xml` file and update the Checkstyle configuration to indicate the file that includes all the suppression filters.

```
...output omitted...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>3.1.1</version>
  <configuration>
    ...output omitted...
    <suppressionsLocation>checkstyle-suppressions.xml</suppressionsLocation>
  ...output omitted...
```

- 3.3. Open the `src/main/java/com/redhat/simple/calculator/BasicCalculator.java` file and update the implementation to pass the code style validations. The file contents should be similar to the following:

```
package com.redhat.simple.calculator;

import java.util.Random;

public final class BasicCalculator {
    public int divide(final int dividend, final int divisor) {
        if (divisor == 0) {
            return Integer.MAX_VALUE;
        } else {
            return dividend / divisor;
        }
    }

    public int subs(final int minuend, final int subtrahend) {
        return minuend - subtrahend;
    }

    public int sum(final int addendA, final int addendB) {
        return addendA + addendB;
    }

    public int multiply(final int multiplicand, final int multiplier) {
        return multiplicand * multiplier;
    }

    public int random() {
        Random r = new Random();

        return r.nextInt();
    }
}
```

Note that the file needs to end with an empty line to pass the style conventions.

- 3.4. Run the Maven `checkstyle:check` goal to verify that the code follows the Sun Code Conventions after the update.

```
[user@host quality-checks]$ ./mvnw clean checkstyle:check
...output omitted...
[INFO] Starting audit...
Audit done.
[INFO] You have 0 Checkstyle violations.
...output omitted...
```

► 4. Detect duplication's in the code of the simple calculator.

- 4.1. Open the `pom.xml` file and add the PMD plugin to the `build.plugins` section.

```

...output omitted...
<build>
  <plugins>
    ...output omitted...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>3.13.0</version>
      <configuration>
        <minimumTokens>10</minimumTokens>
        <printFailingErrors>true</printFailingErrors>
        <linkXRef>false</linkXRef>
      </configuration>
    </plugin>
    ...output omitted...
  </plugins>
</build>
...output omitted...

```

The preceding code adds the PMD plugin, logs errors to console, and sets the minimum of tokens to detect duplicates to 10.



Note

The low value in the minimum tokens to detect duplicates is for educational purposes.

4.2. Run the Maven `pmd:cpd-check` goal to find duplication's in the code.

```

[user@host quality-checks]$ ./mvnw clean pmd:cpd-check
...output omitted...
[ERROR] Failed to execute goal ...output omitted... You have 1 CPD
duplication. ...output omitted...
...output omitted...

```

The copy-paste detector found that the `BasicCalculator.java` and `AdvancedCalculator.java` files are similar. Open the `target/site/cpd.html` file with your browser to view the HTML version of the report generated by the copy-paste detector.

► 5. Apply a refactoring strategy to remove duplicates.

The calculator has two classes with different purposes but they share some methods. To remove duplicates you must follow the Extract Superclass strategy. This strategy creates a shared superclass and moves all common methods to it.

5.1. Create the `src/main/java/com/redhat/simple/calculator/Calculator.java` file with these common methods. The file contents should be similar to the following:

```

package com.redhat.simple.calculator;

public abstract class Calculator {
    public final int divide(final int dividend, final int divisor) {

```

```

        if (divisor == 0) {
            return Integer.MAX_VALUE;
        } else {
            return dividend / divisor;
        }
    }

    public final int subs(final int minuend, final int subtrahend) {
        return minuend - subtrahend;
    }

    public final int sum(final int addendA, final int addendB) {
        return addendA + addendB;
    }

    public final int multiply(final int multiplicand, final int multiplier) {
        return multiplicand * multiplier;
    }
}

```

Note that all the methods in the class are `final` to avoid children classes overriding them. Do not forget to add an empty line at the end of the file to pass the style conventions.

- 5.2. Open the `src/main/java/com/redhat/simple/calculator/BasicCalculator.java` file and make the following changes:

- Extend the `BasicCalculator` class from the `Calculator` abstract class.
- Remove all the common methods.

The updated file should be similar to the following:

```

package com.redhat.simple.calculator;

import java.util.Random;

public final class BasicCalculator extends Calculator {

    public int random() {
        Random r = new Random();

        return r.nextInt();
    }
}

```

- 5.3. Open the `src/main/java/com/redhat/simple/calculator/AdvancedCalculator.java` file and make the following changes:

- Extend the `AdvancedCalculator` class from the `Calculator` abstract class.
- Remove all the common methods.

The updated file should be similar to the following:

```

package com.redhat.simple.calculator;

public final class AdvancedCalculator extends Calculator {
    static final double PI = 3.14;
    ...output omitted...
    static final int GOLD_CUSTOMER_SEGMENT = 3;

    public double circumference(final int r) {
        return 2 * PI * r;
    }

    public boolean isGreaterThan(final int a, final int b) {
        return a > b;
    }

    public double discount(final double saleAmount,
                           final int yearsAsCustomer,
                           final boolean isBusiness) {
        ...output omitted...
    }
}

```

- 5.4. Run the Maven `pmd:cpd-check` goal to verify that after the refactor there are no code duplication's.

```

[user@host quality-checks]$ ./mvnw clean pmd:cpd-check
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...

```

- 5.5. The simple calculator follows the TDD principles. Run the tests to verify that they pass after the refactor.

```

[user@host quality-checks]$ ./mvnw test
...output omitted...
[INFO]
[INFO] Tests run: 21, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...

```

- ▶ 6. Measure the test coverage of the calculator.

- 6.1. Open the `pom.xml` file and add the Jacoco plugin to the `build.plugins` section.

```

...output omitted...
<build>
  <plugins>
    ...output omitted...
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.5</version>
      <executions>
        <execution>
          <id>default-prepare-agent</id>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>
          <id>default-report</id>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  ...output omitted...

```

6.2. Run the Maven `verify` phase to generate the code coverage report.

```

[user@host quality-checks]$ ./mvnw clean verify
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...

```

The `verify` phase executes Jacoco and generates a report in the `target/jacoco.exec` file. You can use the report file in modern IDEs to visualize the code coverage of your code.

6.3. Open the `target/site/jacoco/index.html` file with your browser to view the HTML version of the report generated by Jacoco.

The report indicates the following:

- 91% of instructions covered by tests.
- 95% of decision branches covered by tests.

6.4. Click `com.redhat.simple.calculator` to see the report for each one of the classes of the application.

6.5. Click `Calculator` and notice that the tests are missing the coverage of some logical branches in the `divide` method.

6.6. Click `divide` to see exactly which part of this method is not covered by tests. The report indicates that the `divisor == 0` branch is not covered by tests.

- 6.7. Open the `src/test/java/com/redhat/simple/calculator/BasicCalculatorTest.java` file and add a test for the case of a division by zero. The test should be similar to the following:

```
@Test
void testDivideByZero() {
    assertEquals(Integer.MAX_VALUE, calculator.divide(1, 0));
}
```

- 6.8. Run the Maven `verify` phase to run the tests and generate a new code coverage report, after the inclusion of the new test.

```
[user@host quality-checks]$ ./mvnw clean verify
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 6.9. Open the `target/site/jacoco/index.html` file with your browser to view the HTML version of the report.
- 6.10. Click `com.redhat.simple.calculator` to see the detailed report for each one of the components of the project. Notice how the test coverage for the `Calculator` class increased after adding a test for a missing logical branch. Now the `Calculator` class has a coverage of 100% for the instructions and logical branches.

► 7. Detect methods with a high cyclomatic complexity.

- 7.1. Create a file named `pmd-rulesets.xml` in the root of the project to store all the rules you intend to execute in a PMD run. Add the cyclomatic complexity rule to the rule set. The file contents should be similar to:

```
<?xml version="1.0"?>
<ruleset name="PMD-Rules">
    <description>PMD rules</description>
    <rule ref="category/java/design.xml/CyclomaticComplexity" />
</ruleset>
```

- 7.2. Open the `pom.xml` file and update the PMD configuration that indicates the file, which includes all the rules to execute in PMD.

```
...output omitted...
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-pmd-plugin</artifactId>
    <version>3.13.0</version>
    <configuration>
        ...output omitted...
        <rulesets>
            <ruleset>${pom.basedir}/pmd-rulesets.xml</ruleset>
        </rulesets>
        ...output omitted...
```


- 7.3. Run the Maven `pmd:pmd` goal to execute PMD and generate a report about the cyclomatic complexity of the project.

```
[user@host quality-checks]$ ./mvnw clean pmd:pmd
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 7.4. Open the `target/site/pmd.html` file with your browser to view the HTML version of the report generated by PMD.

The report indicates that the `discount` method of the `AdvancedCalculator` class has a cyclomatic complexity of 11. The cyclomatic complexity affects the maintenance costs and readability of the code.

- 7.5. Open the `src/main/java/com/redhat/simple/calculator/AdvancedCalculator.java` file and refactor the `discount` method. You can extract all the business logic of the `discount` method to smaller methods, to reduce the complexity.

Add a method to extract the logic of calculating the discount for the different customer segments. The method code should be similar to the following:

```
private double discountPerCustomerSegment(final int yearsAsCustomer) {
    double discount;

    switch (yearsAsCustomer) {
        case BRONZE_CUSTOMER_SEGMENT:
            discount = LOW_DISCOUNT_AMOUNT;
            break;
        case SILVER_CUSTOMER_SEGMENT:
            discount = MID_DISCOUNT_AMOUNT;
            break;
        case GOLD_CUSTOMER_SEGMENT:
            discount = HIGH_DISCOUNT_AMOUNT;
            break;
        default:
            discount = VIP_DISCOUNT_AMOUNT;
            break;
    }

    return discount;
}
```

Add a method to extract the logic of calculating the discount for low amount purchases. The method code should be similar to the following:

```
private double discountForLowPurchases(final int yearsAsCustomer,
                                       final boolean isBusiness) {

    if (isBusiness) {
        return LOW_DISCOUNT_AMOUNT;
    }
}
```

```

    if (yearsAsCustomer > GOLD_CUSTOMER_SEGMENT) {
        return MID_DISCOUNT_AMOUNT;
    }

    return LOW_DISCOUNT_AMOUNT;
}

```

Add a method to extract the logic of calculating the discount for high amount purchases. The method code should be similar to the following:

```

private double discountForHighPurchases(final int yearsAsCustomer,
                                       final boolean isBusiness) {

    if (isBusiness) {
        return VIP_DISCOUNT_AMOUNT;
    }

    return discountPerCustomerSegment(yearsAsCustomer);
}

```

Refactor the `discount` method to use the auxiliary methods. The `discount` method should be similar to the following:

```

public double discount(final double saleAmount,
                      final int yearsAsCustomer,
                      final boolean isBusiness) {

    if (saleAmount < MINIMUM_PURCHASE_AMOUNT
        || yearsAsCustomer < BRONZE_CUSTOMER_SEGMENT) {
        return 0;
    }

    if (saleAmount < LOW_PURCHASE_AMOUNT) {
        return discountForLowPurchases(yearsAsCustomer, isBusiness);
    } else if (saleAmount > HIGH_PURCHASE_AMOUNT) {
        return discountForHighPurchases(yearsAsCustomer, isBusiness);
    }

    return 0;
}

```

7.6. Run the Maven `pmd:pmd` goal to execute PMD and generate a new report.

```

[user@host quality-checks]$ ./mvnw clean pmd:pmd
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...

```

7.7. Open the `target/site/pmd.html` file with your browser to view the HTML version of the report.

Notice that, after the refactor, the cyclomatic complexity is below the threshold and PMD can not find problems in the source code.

- 7.8. Run the Maven `verify` phase to run the tests and generate a new code coverage report.

```
[user@host quality-checks]$ ./mvnw clean verify
...output omitted...
[INFO] Tests run: 22, Failures: 0, Errors: 0, Skipped: 0
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

The tests continue passing after the refactor.

- 7.9. Open the `target/site/jacoco/index.html` file with your browser to view the test coverage report.
- 7.10. Click `com.redhat.simple.calculator`, then `AdvancedCalculator`, and finally `discount`. Notice that, after the refactor, the tests continue covering all the new code.

This concludes the guided exercise.

► Lab

Building Applications with Test-driven Development

In this lab, you will use TDD practices to add a new feature to a simple calculator application. Adding automatic code analysis tools will raise potential issues that you will address by using testing and development best practices.

You will find the solution files for this lab in the `solutions` branch of the `D0400-apps` repository, within the `calculator-monolith` folder.

Outcomes

You should be able to:

- Apply TDD practices to develop a new features in a simple application
- Implement development best practices to improve code quality
- Enable code quality automatic checks

Before You Begin

If not done before, fork the <https://github.com/RedHatTraining/D0400-apps> repository into your own GitHub account, clone the fork locally, and move to the `main` branch. Use the `calculator-monolith` folder containing the calculator application.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

```
[user@host D0400]$ cd D0400-apps/calculator-monolith
[user@host shipping-calculator]$ git checkout main
...output omitted...
```

This application exposes a single endpoint `/solver/<equation>` that tries to solve the provided equation. In its current state, the application can only solve sums and subtractions of decimal numbers. For example, the endpoint `/solve/4.02+8.0-1.1` returns the string `10.92`.

You must extend the application for it to be able to use the multiplication operation in formulas. For example, the endpoint `/solve/4*8-1` must return the string `31.0`.

Be aware of operation priorities: multiplications must always be solved before sums or subtractions.

You must also enable the PMD and Checkstyle tools to ensure the code quality of the application. Disable some Checkstyle rules to adapt the analysis to development tools and conventions. Fix any issues raised by the code analysis tools by using development best practices and principles.

Instructions

- Review the existing application. Open the application in the `DO400-apps/calculator-monoLith` folder and review its contents.
 - The `SolverResource` class, in the `src/main/java/com/redhat/training/SolverResource.java` file, acts as the controller, by exposing the REST endpoints and connecting these endpoints with their corresponding operations.
 - The `operation` folder contains the implementation of arithmetic operations, which follow the `Strategy` design pattern [https://en.wikipedia.org/wiki/strategy_pattern]. This pattern allows you to decouple your code from different implementations of a task or algorithm. In this case, the `Operation` interface in the `src/main/java/com/redhat/training/operation/Operation.java` file defines the abstract task: an arithmetic operation. The specific operations, such as the `Add` and `Subtract` classes, are concrete strategies.
 - The application embraces the `Inversion Of Control` principle, by delegating the control of dependencies to the `Java Contexts and Dependency Injection (CDI)` framework. The many `@Inject` annotations declare the injection points where the CDI context must inject dependencies. The `@ApplicationScoped` annotations declare objects managed by the CDI context.
 - The code at the `src/test/java/com/redhat/training/operation` folder contains tests for individual operation classes. The `src/test/java/com/redhat/training/SolverTest.java` class unit tests the application endpoint class. The `src/test/java/com/redhat/training/SolverIT.java` class defines an integration test, which validates the correct behavior of endpoints and operation classes together.
- Implement the new `multiply` operation for the calculator application. Use as many TDD cycles as appropriate.

You must at least create a unit test for the multiplication operation, and an integration test that verifies the multiplication operation in relation to the equation endpoint.

You can copy and modify the `src/main/java/com/redhat/training/operation/Add.java` or `Subtract.java` classes and just modify the `REGEX` and `OPERATOR` fields. Add the new operation to the list of operations in the `com.redhat.training.SolverResource.buildOperationList` method.
- Enable the `PMD` and `Checkstyle` code quality automated tools. `PMD` must run the `check` and `cpd-check` goals in the `validate` phase. Set to 25 the minimum number of duplicated tokens that raise a `CPD` violation.

Attach `Checkstyle` to the same `validate` phase, but disable the following rules:

 - `Javadoc`, as the current legacy codebase is missing comments almost everywhere.
 - `LineLength`, as per local development conventions.
 - `DesignForExtension` and `VisibilityModifier`, as they go against Quarkus proxy features.
- Fix code quality issues raised by the code analysis tools. Execute the `./mvnw verify` command and verify that execution fails due to `Checkstyle` violations. Many `Checkstyle` violations can be fixed by simply applying the default formatter to files, however, others require deeper code refactoring.
- The `SolverResource` class breaks the `SRP (Single Responsibility Principle)`. It acts both as the entry point for calculations and as an operation registry.

Apply development best practices and principles to fix this violation.

This concludes the lab.

► Solution

Building Applications with Test-driven Development

In this lab, you will use TDD practices to add a new feature to a simple calculator application. Adding automatic code analysis tools will raise potential issues that you will address by using testing and development best practices.

You will find the solution files for this lab in the `solutions` branch of the `D0400-apps` repository, within the `calculator-monolith` folder.

Outcomes

You should be able to:

- Apply TDD practices to develop a new features in a simple application
- Implement development best practices to improve code quality
- Enable code quality automatic checks

Before You Begin

If not done before, fork the <https://github.com/RedHatTraining/D0400-apps> repository into your own GitHub account, clone the fork locally, and move to the `main` branch. Use the `calculator-monolith` folder containing the calculator application.



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

```
[user@host D0400]$ cd D0400-apps/calculator-monolith
[user@host shipping-calculator]$ git checkout main
...output omitted...
```

This application exposes a single endpoint `/solver/<equation>` that tries to solve the provided equation. In its current state, the application can only solve sums and subtractions of decimal numbers. For example, the endpoint `/solve/4.02+8.0-1.1` returns the string `10.92`.

You must extend the application for it to be able to use the multiplication operation in formulas. For example, the endpoint `/solve/4*8-1` must return the string `31.0`.

Be aware of operation priorities: multiplications must always be solved before sums or subtractions.

You must also enable the PMD and Checkstyle tools to ensure the code quality of the application. Disable some Checkstyle rules to adapt the analysis to development tools and conventions. Fix any issues raised by the code analysis tools by using development best practices and principles.

Instructions

1. Review the existing application. Open the application in the `D0400-apps/calculator-monoLith` folder and review its contents.
 - The `SolverResource` class, in the `src/main/java/com/redhat/training/SolverResource.java` file, acts as the controller, by exposing the REST endpoints and connecting these endpoints with their corresponding operations.
 - The `operation` folder contains the implementation of arithmetic operations, which follow the `Strategy` design pattern [https://en.wikipedia.org/wiki/strategy_pattern]. This pattern allows you to decouple your code from different implementations of a task or algorithm. In this case, the `Operation` interface in the `src/main/java/com/redhat/training/operation/Operation.java` file defines the abstract task: an arithmetic operation. The specific operations, such as the `Add` and `Subtract` classes, are concrete strategies.
 - The application embraces the `Inversion Of Control` principle, by delegating the control of dependencies to the `Java Contexts and Dependency Injection (CDI)` framework. The many `@Inject` annotations declare the injection points where the CDI context must inject dependencies. The `@ApplicationScoped` annotations declare objects managed by the CDI context.
 - The code at the `src/test/java/com/redhat/training/operation` folder contains tests for individual operation classes. The `src/test/java/com/redhat/training/SolverTest.java` class unit tests the application endpoint class. The `src/test/java/com/redhat/training/SolverIT.java` class defines an integration test, which validates the correct behavior of endpoints and operation classes together.
2. Implement the new `multiply` operation for the calculator application. Use as many TDD cycles as appropriate.

You must at least create a unit test for the multiplication operation, and an integration test that verifies the multiplication operation in relation to the equation endpoint.

You can copy and modify the `src/main/java/com/redhat/training/operation/Add.java` or `Subtract.java` classes and just modify the `REGEX` and `OPERATOR` fields. Add the new operation to the list of operations in the `com.redhat.training.SolverResource.buildOperationList` method.

 - 2.1. Create a unit test for the implementation of the `Multiply` operation.

You can create from scratch or copy and modify the `src/test/java/com/redhat/training/operation/AddTest.java` file to another file in the same folder, for example `src/test/java/com/redhat/training/operation/MultiplyTest.java`.

```
@QuarkusTest
@Tag("unit")
public class MultiplyTest {

    @Inject
    Multiply multiply;

    @Test
    public void simple_multiplication() {
        assertEquals(multiply.apply("4*5"), 20);
    }
}
```



```

@Test
public void unparseable_operation() {
    assertNull(multiply.apply("4-5"));
}
}

```

Execute the test by using the `./mvnw verify` command. The new test must fail, because the multiply operation is still not present.

2.2. Develop the minimum set of code that satisfies the unit tests.

Create the `Multiply` operation class by making a copy of another operation and replacing the `REGEX` and the `OPERATOR` attributes. For example, copy the `src/main/java/com/redhat/training/operation/Add.java` to a file named `src/main/java/com/redhat/training/operation/Multiply.java` and rename the class inside to `Multiply`, and then edit the attributes as follows:

```

@ApplicationScoped
public final class Multiply implements Operation {
    private static final String REGEX = "(.)\\*(.)";
    private static final BinaryOperator<Float> OPERATOR = (lhs, rhs) -> lhs * rhs;

    public Multiply() {
        super();
    }

    @Override
    public Float apply(final String equation) {
        return solveGroups(equation).stream().reduce(OPERATOR).orElse(null);
    }

    private List<Float> solveGroups(final String equation) {
        Matcher matcher = Pattern.compile(REGEX).matcher(equation);
        if (matcher.matches()) {
            List<Float> result = new ArrayList<>(matcher.groupCount());
            for (int groupNum = 1; groupNum <= matcher.groupCount(); groupNum++) {
                result.add(solve(matcher.group(groupNum)));
            }
            return result;
        } else {
            return Collections.emptyList();
        }
    }

    @Inject
    SolverService solverService;

    private Float solve(final String equation) {
        return solverService.solve(equation);
    }
}

```

Execute the test by using the `./mvnw verify` command. The new test must run successfully.

- 2.3. Perform any needed refactor and close the TDD cycle. The new code is a copy of an existing one, so almost no refactor applies.
- 2.4. Update the unit and integration tests of the `SolverTest` class to cover the addition of the new multiply operation into the existing endpoints.

Modify the `src/test/java/com/redhat/training/SolverTest.java` class to include unit tests for the `Solver` class. For example, add the following two methods anywhere in the `SolverTest` class.

```
@QuarkusTest
@Tag("unit")
public class SolverTest {
    ...output omitted...

    @Test
    public void solve_multiply() {
        assertEquals(solverService.solve("5*3"), 15);
    }

    @Test
    public void solve_multiplication_overprioritize_addition() {
        assertEquals(solverService.solve("10-5*3+2"), -7);
    }
    ...output omitted...
```

In relation to the integration tests, update the `src/test/java/com/redhat/training/SolverIT.java` class to add two tests. The first one, a test which validates a single multiplication endpoint. The second one, a test which validates that the multiplication integrates with other operations honouring the resolution order.

```
@QuarkusTest
@TestHTTPEndpoint(SolverService.class)
@Tag("integration")
public class SolverIT {
    ...output omitted...

    @Test
    public void solve_multiply() {
        expectEquationSolution("4*2", "8.0");
    }

    @Test
    public void solve_composed_multiply() {
        expectEquationSolution("4+2*3", "10.0");
    }
    ...output omitted...
```

Execute the test by using the `./mvnw verify` command. The tests must fail, because the new `Multiply` operation is still not integrated in the `SolverResource` class.

- 2.5. Develop the minimum set of code that satisfies the tests.
- Update the `src/main/java/com/redhat/training/SolverResource.java` class. Import the `com.redhat.training.operation.Multiply` class, and inject an instance of that class into the `SolverResource` class. Add the `multiply` operation to the operation list in the `buildOperationList` method. It is important to

add the operation between the `add` and `identity` operations, so the evaluation order is appropriate.

```
...output omitted...

import com.redhat.training.operation.Add;
import com.redhat.training.operation.Identity;
import com.redhat.training.operation.Multiply;
import com.redhat.training.operation.Operation;
import com.redhat.training.operation.Subtract;
import com.redhat.training.service.SolverService;

...output omitted...
public final class SolverResource implements SolverService {
    private static final Logger LOG =
        LoggerFactory.getLogger(SolverResource.class);
    ...output omitted...

    @Inject
    Multiply multiply;

    ...output omitted...
    @PostConstruct
    public void buildOperationList() {
        operations = List.of(subtract, add, multiply, identity);
    }
}
```

Verify the implementation with the `./mvnw verify` command. All tests must pass. Otherwise, review the code and perform the needed fixes.

- 2.6. Refactor code as needed and validate that tests are passing now. Refactoring depends greatly on how the code was developed. Make sure your code follows both testing and development best practices and refactor as needed.

You do not need to add more features, so the TDD cycle ends here.

3. Enable the PDM and Checkstyle code quality automated tools. PMD must run the check and cpd-check goals in the `validate` phase. Set to 25 the minimum number of duplicated tokens that raise a CPD violation.

Attach Checkstyle to the same `validate` phase, but disable the following rules:

- Javadoc, as the current legacy codebase is missing comments almost everywhere.
- LineLength, as per local development conventions.
- DesignForExtension and VisibilityModifier, as they go against Quarkus proxy features.

- 3.1. Add the PMD plugin to the `pom.xml` file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-pmd-plugin</artifactId>
  <version>3.14.0</version>
  <configuration>
    <minimumTokens>25</minimumTokens>
    <printFailingErrors>true</printFailingErrors>
    <linkXRef>false</linkXRef>
  </configuration>
</plugin>
```

```

</configuration>
<executions>
  <execution>
    <id>validate</id>
    <phase>validate</phase>
    <goals>
      <goal>check</goal>
      <goal>cpd-check</goal>
    </goals>
  </execution>
</executions>
</plugin>

```

3.2. Add the checkstyle plugin to the pom.xml file.

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>3.1.1</version>
  <configuration>
    <consoleOutput>true</consoleOutput>
    <failsOnError>false</failsOnError>
    <linkXRef>false</linkXRef>
    <suppressionsLocation>checkstyle-suppressions.xml</suppressionsLocation>
  </configuration>
  <executions>
    <execution>
      <id>validate</id>
      <phase>validate</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

3.3. To specify rule suppressions, create the checkstyle-suppressions.xml file, in the application root folder, with the following content:

```

<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
  "-//Checkstyle//DTD SuppressionFilter Configuration 1.2//EN"
  "https://checkstyle.org/dtds/suppressions_1_2.dtd">
<suppressions>
  <suppress checks="Javadoc" files=".java" />
  <suppress checks="LineLength" files=".java" />
  <suppress checks="VisibilityModifier" files=".java" />
  <suppress checks="DesignForExtension" files=".java" />
</suppressions>

```

The files pattern can be different in your implementation but make sure it covers all Java files.

4. Fix code quality issues raised by the code analysis tools. Execute the `./mvnw verify` command and verify that execution fails due to Checkstyle violations. Many Checkstyle violations can be fixed by simply applying the default formatter to files, however, others require deeper code refactoring.
 - 4.1. Remove duplicated code found by CPD. PMD detects through CPD that the three binary operations share a chunk of code.

```
[INFO] CPD Failure: Found 30 lines of duplicated code at locations:
[INFO]     .../src/main/java/com/redhat/training/operation/Add.java line 20
[INFO]     .../src/main/java/com/redhat/training/operation/Multiply.java line 20
[INFO]     .../src/main/java/com/redhat/training/operation/Subtract.java line 20
```

This is a clear indicator of a violation of the DRY (Don't Repeat Yourself) principle.

To solve this violation, you can create a new parent class named `BinaryOperation`, extend all binary operations from that class, and move the duplicated methods to that class.

Create a new `BinaryOperation.java` file in the `src/main/java/com/redhat/training/operation` folder, and perform the following edits:

- The `BinaryOperation` class must be abstract and must implement the `Operation` interface.
- The `Add`, `Subtract` and `Multiply` classes must extend the `BinaryOperation` class. They must not implement the `Operation` interface explicitly.
- Move all methods but the class constructor from the `Add`, `Subtract`, and `Multiply` classes to the `BinaryOperation` class.
- Do not move the `REGEX` and `OPERATOR` constants to the `BinaryOperation` class. Instead, create a `regex` and an `operator` field in the `BinaryOperation` class and add them as parameters to the constructor method.
- Remove unused imports from operation classes.

Note the `Identity` operator does not follow the same structure (does not appear as duplicate code), so it must not be modified.

The `BinaryOperation` class must look like the following:

```
package com.redhat.training.operation;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.function.BinaryOperator;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.inject.Inject;

import com.redhat.training.service.SolverService;

public abstract class BinaryOperation implements Operation {

    private final BinaryOperator<Float> operator;
    private final String regex;
```

```

    public BinaryOperation(final BinaryOperator<Float> operatorParam, final String
regexParam) {
        this.operator = operatorParam;
        this.regex = regexParam;
    }

    public Float apply(final String equation) {
        return solveGroups(equation).stream().reduce(operator).orElse(null);
    }

    private List<Float> solveGroups(final String equation) {
        Matcher matcher = Pattern.compile(regex).matcher(equation);
        if (matcher.matches()) {
            List<Float> result = new ArrayList<>(matcher.groupCount());
            for (int groupNum = 1; groupNum <= matcher.groupCount(); groupNum++) {
                result.add(solve(matcher.group(groupNum)));
            }
            return result;
        } else {
            return Collections.emptyList();
        }
    }

    @Inject
    SolverService solverService;

    private Float solve(final String equation) {
        return solverService.solve(equation);
    }
}

```

The Add, Subtract and Multiply classes must contain only the required imports, the constants, and the constructor.

```

package com.redhat.training.operation;

import java.util.function.BinaryOperator;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public final class Add extends BinaryOperation {
    private static final String REGEX = "(.+)\\+(.+)";
    private static final BinaryOperator<Float> OPERATOR = (lhs, rhs) -> lhs + rhs;

    public Add() {
        super(OPERATOR, REGEX);
    }
}

```

Re-run the `./mvnw verify` command to validate the Checkstyle violations are gone and the test pass.

5. The `SolverResource` class breaks the SRP (Single Responsibility Principle). It acts both as the entry point for calculations and as an operation registry.

Apply development best practices and principles to fix this violation.

- 5.1. Create a class that acts as a registry of available operations, to extract this responsibility from the `SolverResource` class.

Create the `com.redhat.training.OperationRegistry` class, in a new `src/main/java/com/redhat/training/OperationRegistry.java` file. This class will act as the registry for allowed operations. Make sure this class is managed by the CDI framework by using the `@ApplicationScoped` annotation.

Move the `buildOperationList`, the list of operations, and all the `@Inject` operation fields from the `SolverResource` class to the new `OperationRegistry` class. Create a getter method for the operations list field named `getOperations`.

```
package com.redhat.training;

import java.util.List;

import javax.annotation.PostConstruct;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import com.redhat.training.operation.Add;
import com.redhat.training.operation.Identity;
import com.redhat.training.operation.Multiply;
import com.redhat.training.operation.Operation;
import com.redhat.training.operation.Subtract;

@ApplicationScoped
public class OperationRegistry {

    @Inject
    Add add;

    @Inject
    Subtract subtract;

    @Inject
    Multiply multiply;

    @Inject
    Identity identity;

    private List<Operation> operations;

    @PostConstruct
    protected void buildOperationList() {
        setOperations(List.of(subtract, add, multiply, identity));
    }

    public List<Operation> getOperations() {
        return operations;
    }

    private void setOperations(final List<Operation> operationsParam) {
```

```

        this.operations = operationsParam;
    }
}

```

Finally, inject the new registry class into the `SolverResource` class, and replace the access to the `operation` field with a reference getter method created in the previous step. Remember to format the file and remove unused imports.

The class body should look like the following:

```

public final class SolverResource implements SolverService {
    private static final Logger LOG =
        LoggerFactory.getLogger(SolverResource.class);

    @Inject
    OperationRegistry operations;

    @Override
    public Float solve(@PathParam("equation") final String equation) {
        LOG.info("Solving '{}'", equation);
        for (Operation operation : operations.getOperations()) {
            Float result = operation.apply(equation);
            if (result != null) {
                LOG.info("Solved '{}' = {}", equation, result);
                return result;
            }
        }

        throw new WebApplicationException(
            Response.status(Response.Status.BAD_REQUEST).entity("Unable to
            parse: " + equation).build());
    }
}

```

Execute the `./mvnw verify` command. Solve any violations that arise and repeat the cycle until all tests execute successfully.



Note

The approach described in this step is sometimes called the *registry pattern*. This pattern defines a central object, the registry, responsible for storing collections of other objects.

In this case, the `OperationRegistry` maintains the available operations, allowing the `SolverResource` class to meet the Single responsibility and Open-Closed principles.

This concludes the lab.

Summary

In this chapter, you learned:

- How to ease development by starting with test-driven development, which focuses on testing small pieces of code.
- Best practices and development patterns to avoid common pitfalls.
- Tools to ensure that Code Quality meets expectations.

Chapter 5

Authoring Pipelines

Goal

Create basic pipelines to run Jenkins jobs.

Objectives

- Create and run a basic Jenkins pipeline.
- Incorporate shell scripts into Jenkins pipelines.
- Create checkpoints, parallel executions, and conditionals for more advanced pipeline behavior.

Sections

- Building a Basic Declarative Pipeline (and Guided Exercise)
- Creating Scripted Pipelines (and Guided Exercise)
- Controlling Step Execution (and Guided Exercise)

Lab

Authoring Pipelines

Building a Basic Declarative Pipeline

Objectives

After completing this section, you should be able to create and run a basic Jenkins pipeline.

Describing Essential Jenkins Concepts

As we covered earlier, *Defining CI/CD and DevOps* lists automation as one of the core concepts of DevOps and CI/CD. Jenkins is a popular open source tool that helps automate building, testing, and deploying applications.

The following concepts are essential for using and managing Jenkins.

Project

A logical way to describe a workflow that Jenkins should perform, such as building an application. This description includes the definition and configuration of the automated tasks.

Job is a deprecated term and is synonymous with *Project*.

Build

The result of a single execution of a project, including its runtime logs and output artifacts.

This term comes from Jenkins' origins as a software build server.

Step

A single task to perform inside a project.

Stage

A subset of steps with a common objective.

Workspace

The working directory where Jenkins executes projects. In this folder, project executions store data that is either temporary, or reused between multiple project executions of the same project.

Node

A machine, or a container capable of executing projects.

Jenkinsfile

A *Jenkinsfile* is a file including a set of instructions for automating project tasks. This file is usually checked in the source control system.

Build Statuses

The execution of the project ends with the assignment of a state to the build. This status helps you understand the condition in which the project execution ended. A build can end in one of the following states.

Aborted

The execution was interrupted by a user, or a time-out.

Failed

The execution ended abruptly because of some fatal error. For example, when an application does not compile.

Successful

The execution finished without fatal errors.

Stable

The execution was successful and without non-fatal errors.

Unstable

The execution finished successfully but with one or more non-fatal errors. For example, when a pipeline execution ends successfully, but the style check stage failed.

Writing a Declarative Pipeline

The Jenkins *Pipeline* is a suite of plug-ins that extend the Jenkins core functionalities. This suite offers a way to define CI/CD pipelines into Jenkins. With the *Pipeline* suite enabled, you can create projects with the pipeline project type.

**Note**

The Jenkins template available in Red Hat OpenShift includes the *Pipeline* suite installed.

A declarative pipeline is composed of blocks, sections, directives, steps, or assignment statements. The following section describes some of the most frequently used elements of a declarative pipeline.

Declarative pipelines must start with a `pipeline` block, and enclose the pipeline definition within this block. The following example illustrates the use of the `pipeline` block.

```
pipeline {
  agent ...output omitted...
  stages {
    ...output omitted...
  }
  post {
    ...output omitted...
  }
}
```

Declarative Pipeline Sections

A section is a segment of a pipeline script that contains one or more directives, or steps. The declarative syntax supports four different sections.

agent

The `agent` section defines conditions to select the machine or container that will execute the pipeline. For example, use `agent { label { 'my_label' } }` to declare that the pipeline must run on a Jenkins agent containing the `my_label` label. Other allowed keywords are `any`, `none`, `node`, or `docker`.

You can override the agent at a per stage level.

stages

The `stages` section defines one or more stages to execute in the pipeline. A stage is an aggregation of steps sharing a common purpose or configuration.

The `stages` section requires one or more `stage` directives: `stages { stage { ... } stage { ... } }`

steps

The `steps` section specifies a list of tasks to execute. The `steps { echo 'Hello World' }` section has a single `echo` task, which prints the message in the pipeline agent.

You can only use steps that are compatible with the Pipeline plug-in.

post

The `post` section defines additional tasks to execute upon the completion of a pipeline, or a stage. The execution of the tasks only occurs when conditions are met. Conditions are expressed with keywords such as `always`, `failure`, `success`, or `cleanup`. For example, you can use the `post` section to send a notification via chat or email when the pipeline fails. The following example highlights the sections involved in a minimal declarative pipeline.

```
pipeline {
  agent any ❶
  stages { ❷
    stage('Hello') {
      steps { ❸
        echo 'Hello World!'
      }
    }
  }
}
```

- ❶ Specifies that the pipeline should run on any available agent.
- ❷ Defines the list of stages. This pipeline only has one stage, named *Hello*.
- ❸ Defines the list of steps to execute in the *Hello* stage. The *Hello* stage only has the `echo` step.



Important

The pipeline block requires at least one `agent`, and one `stages` section.

Declarative Pipeline Directives

The declarative syntax supports multiple directives, and `stage` is one of the most important.

The `stage` directive defines a list of steps with a common goal. This directive is only allowed inside the `stages` section, and has only one mandatory parameter with the name of the stage. The following example highlights the directives involved in a minimal declarative pipeline.

```
pipeline {
  agent any
  stages {
    stage('Hello') { ❶
      steps {
        echo 'Hello World!'
      }
    }
  }
}
```

```

    }
  }
}

```

- ❶ Defines a stage named *Hello*.

Other directives supported in the declarative syntax are: `environment`, `options`, `parameters`, `triggers`, `tools`, `input`, and `when`. Most of these directives will be covered later in the course.

Declarative Pipeline Steps

In Jenkins, tasks are functions defined by the plug-ins. You can integrate those functions in your pipeline scripts. Some of the most used steps are the following:

echo

Sends a message to the standard output. It requires one mandatory string parameter with the message to output.

git

Clones a specified repository. It requires a string parameter with the repository. The optional string `branch` parameter specifies a branch name. The string `url` parameter specifies the git repository.

sh

Executes a shell script. It requires a mandatory string parameter with the script or command to execute.



Note

You can use both single quotes, and double quotes for denoting strings. The main difference between them is that double quotes allow string interpolation.

Use triple single quotes, or triple double quotes for multiline strings.

The following example highlights the steps involved in a minimal declarative pipeline.

```

pipeline {
  agent any
  stages {
    stage('Hello') {
      steps {
        echo 'Hello World!' ❶
      }
    }
  }
}

```

- ❶ Sends the string *Hello World!* to the standard output.

Managing Jenkins

The Jenkins web UI is a web application that enables managing all aspects of a Jenkins server. A standard Jenkins setup provides automation capabilities and is GUI-oriented.

The main elements of the Jenkins web UI are:

- The initial page, which is called the Jenkins **Dashboard**. From there you can navigate to projects and folders.
- A bread crumb trail, which shows your context within the UI. The **Jenkins** link always returns you to the Jenkins **Dashboard** page.
- A left navigation pane with links. These links change according to the context within the UI.

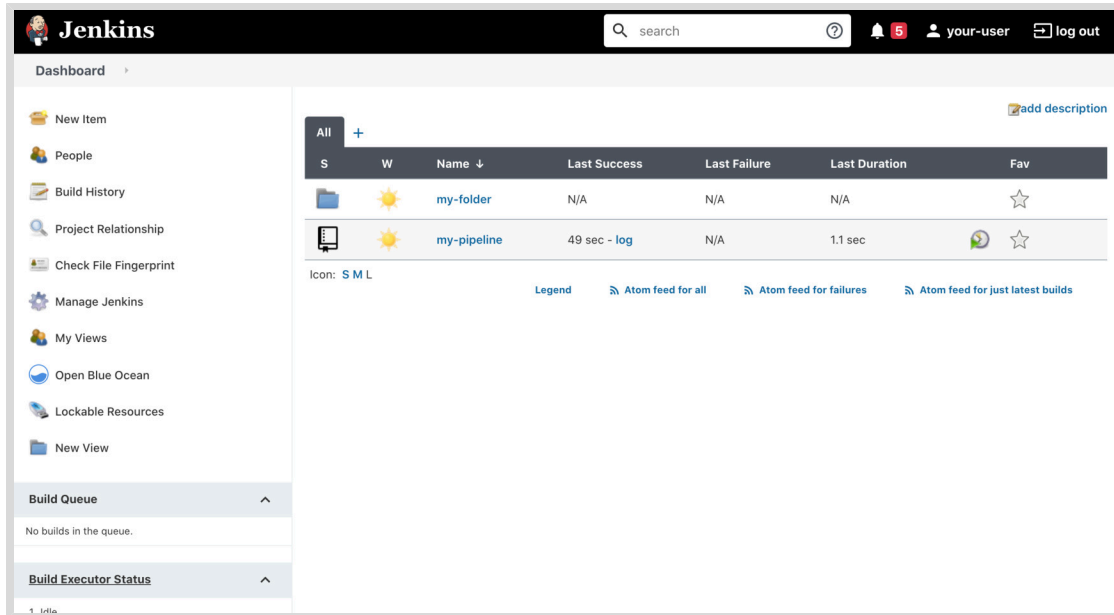


Figure 5.1: Jenkins dashboard

The right pane of the Jenkins **Dashboard** page shows current folders and projects. Clicking on a folder, redirects you to the folder details page; where you see its nested projects and folders. Clicking on a project, redirects you to the project details page; where you see a list of builds for that project.

**Note**

Jenkins also provides a Java-based CLI, and a REST API, which allows limited management of a Jenkins server.

Creating a Declarative Pipeline

The **New Item** link in the Jenkins **Dashboard** page allows you to create different types of projects and other Jenkins configuration items, such as folders to group projects. The most common project types to create pipelines are:

Pipeline

Runs a pipeline taking as input a single branch from a version control system repository. When you are creating a pipeline, you can type your pipeline script in the **Definition** field, located in the **Pipeline** area.

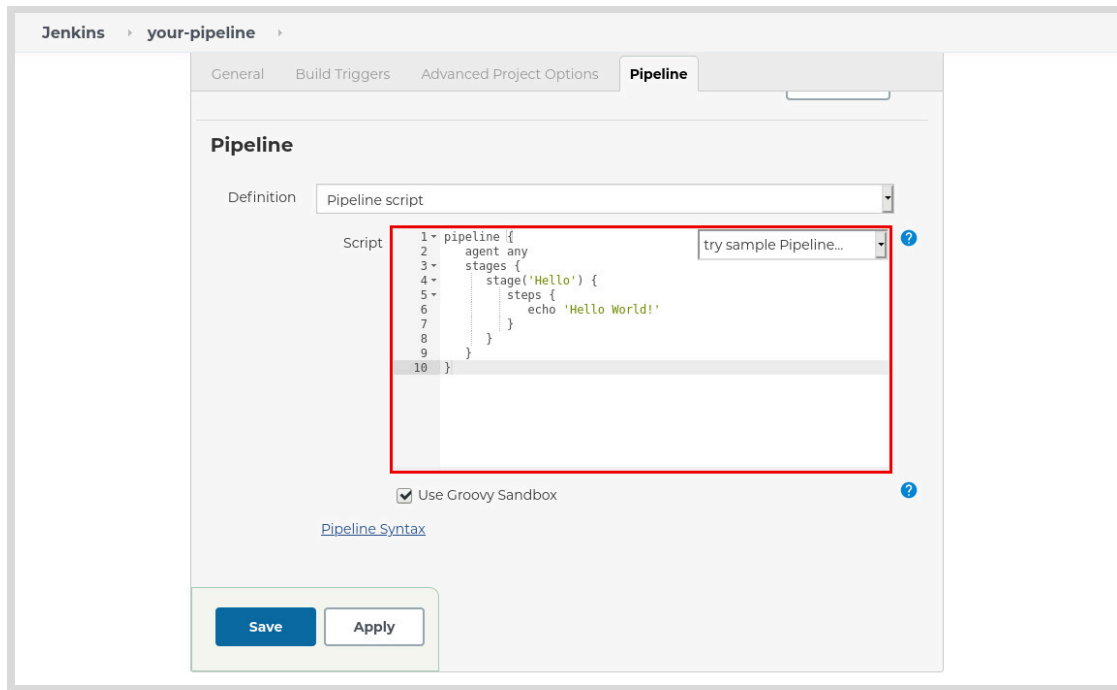


Figure 5.2: Pipeline creation

Multibranch pipeline

Automatically creates new projects when new branches are detected in a version control system repository. All these projects share the same pipeline definition, which must be flexible enough to avoid conflicts between pipeline executions in different branches.

The pipeline configuration settings depend on the type of project, and the installed plug-ins.

Managing a Pipeline

When you click a project in the **Dashboard** page, Jenkins redirects you to the project details page. The main elements of the Jenkins web UI in this page are:

- A left navigation pane with actions related to the current pipeline.
- A **Build History** area, located in the left pane. This area displays all the builds of a project in a chronological order.
- The **Stage View** area, located in the right pane. This area shows the status of the stages in each one of the pipeline executions. Each stage is colored green or red depending on its success (green) or failed (red) status. You can click each stage to see its logs.

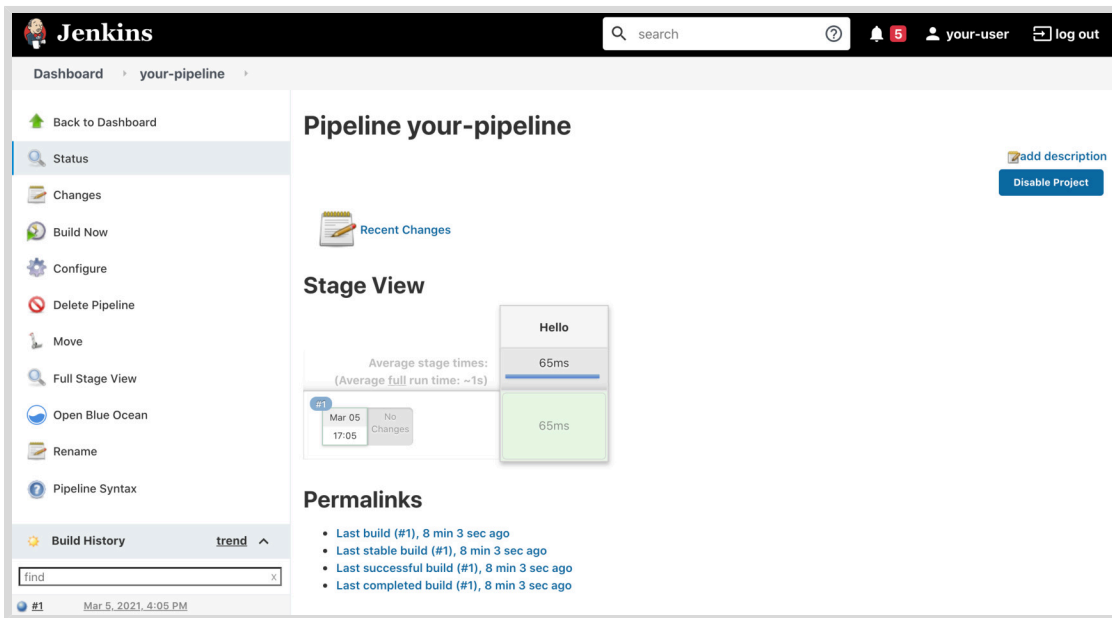


Figure 5.3: Project details page

From the left navigation page you can:

- Start a pipeline execution by clicking **Build Now**.
- Modify the current pipeline setup by clicking **Configure**.
- Delete the pipeline by clicking **Delete Pipeline**.

Managing a Build

When you click a build in the **Build History** area, or in the **Stage View** area, Jenkins redirects you to the build details page. In this page you can:

- See the build log by clicking **Console Output**.
- Delete the current build by clicking **Delete Build**.
- See a detailed report about the execution times by clicking **Timings**.
- Execute the pipeline again with a modified script by clicking **Replay**.
- See a detailed view of the pipeline execution by clicking **Pipeline Steps**.



References

Glossary

<https://www.jenkins.io/doc/book/glossary/>

Remote Access API

<https://www.jenkins.io/doc/book/using/remote-access-api/>

Java Client API

<https://github.com/jenkinsci/java-client-api>

Pipeline Steps Reference

<https://www.jenkins.io/doc/pipeline/steps/>

Pipeline: Nodes and Processes

<https://www.jenkins.io/doc/pipeline/steps/workflow-durable-task-step/>

► Guided Exercise

Building a Basic Declarative Pipeline

In this exercise you will establish and execute a declarative pipeline by using Jenkins. You will also view output from executing the pipeline.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (`) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to create a basic declarative pipeline and navigate the Jenkins interface.

Before You Begin

To perform this exercise, ensure you have:

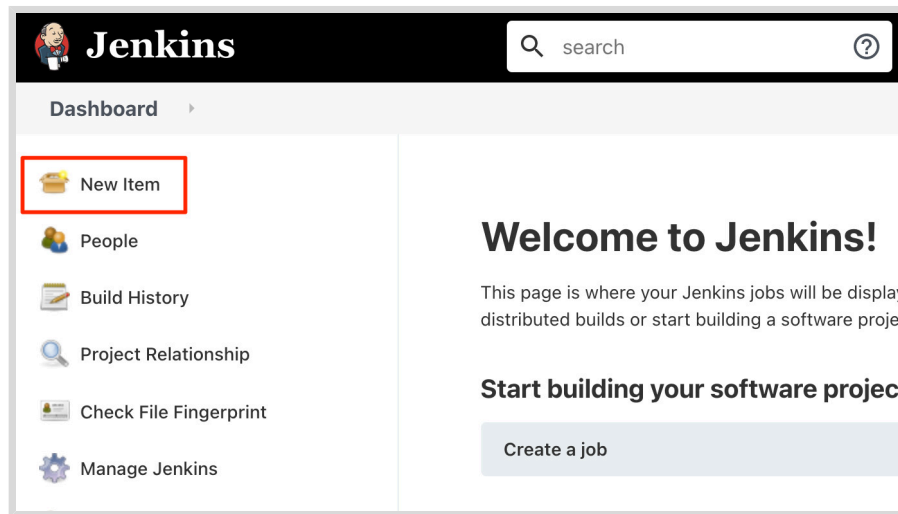
- Access to a Jenkins server
- A web browser such as Firefox or Chrome

Instructions

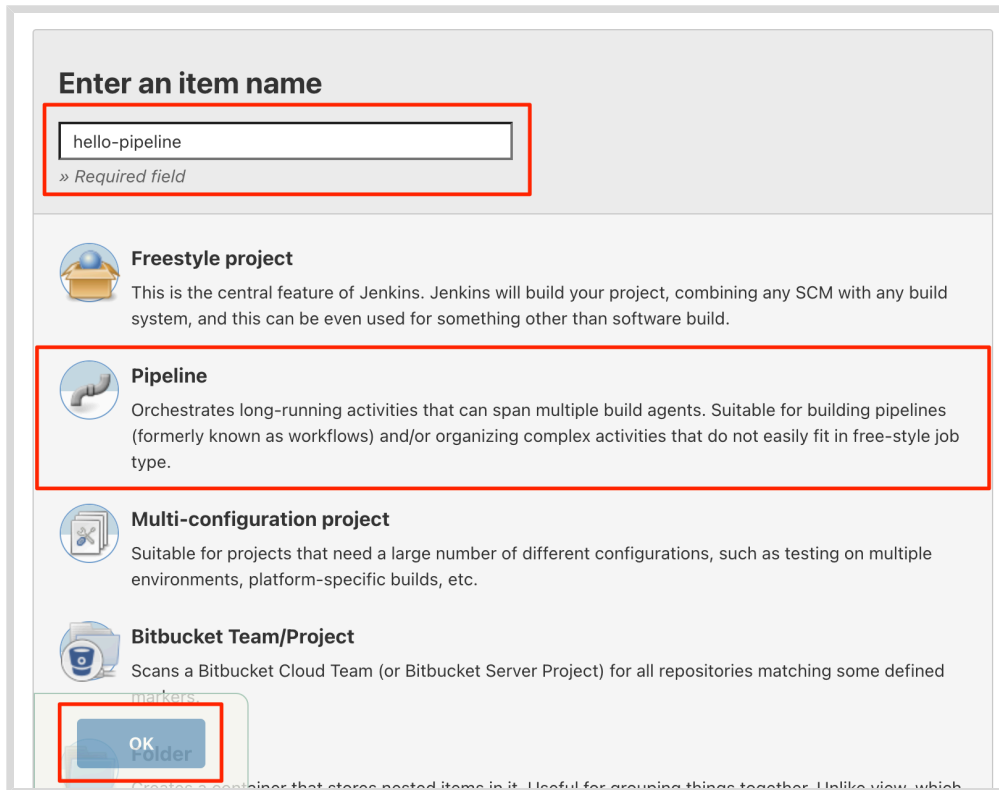
- 1. Create an example single-stage pipeline.
 - 1.1. Open a command-line terminal. Log in to OpenShift with the OpenShift CLI and get the URL of your Jenkins instance. Replace `RHT_OCP4_DEV_USER`, `RHT_OCP4_DEV_PASSWORD`, and `RHT_OCP4_MASTER_API` with the values provided in the **Lab Environment** tab of the course's online learning website.

```
[user@host D0400]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
[user@host D0400]$ oc get route \
-n RHT_OCP4_DEV_USER-jenkins jenkins \
-o jsonpath='{\"http://\"}{.spec.host}{\"\\n\"}'
http://jenkins-your-user-jenkins.apps.cluster.example.com
```

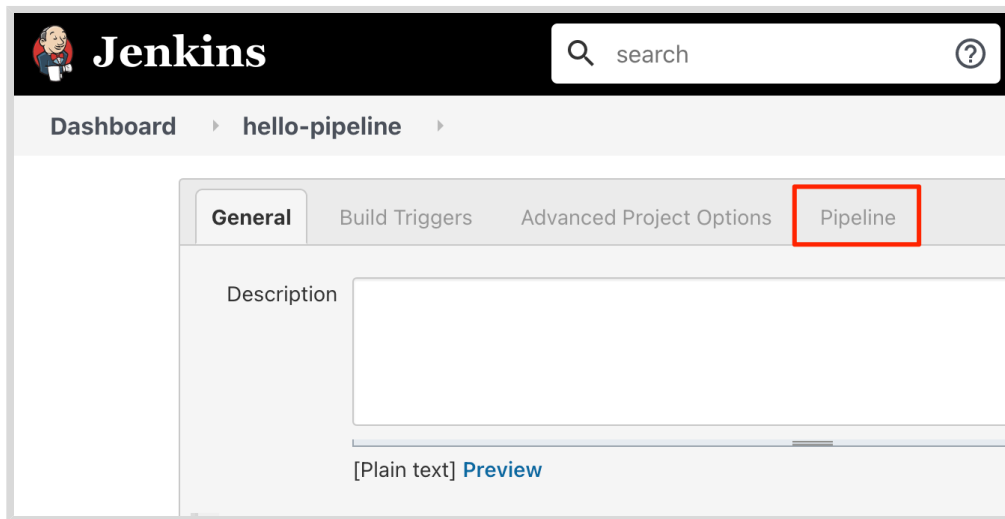
- 1.2. Open your web browser and navigate to the Jenkins URL.
- 1.3. Click **New Item** in the left navigation pane.



1.4. Enter hello-pipeline as the item name. Click Pipeline and then click OK.



1.5. Click the Pipeline tab to scroll down to the Pipeline section.



1.6. Enter the following code in the **Script** field:

```
pipeline {
  agent any

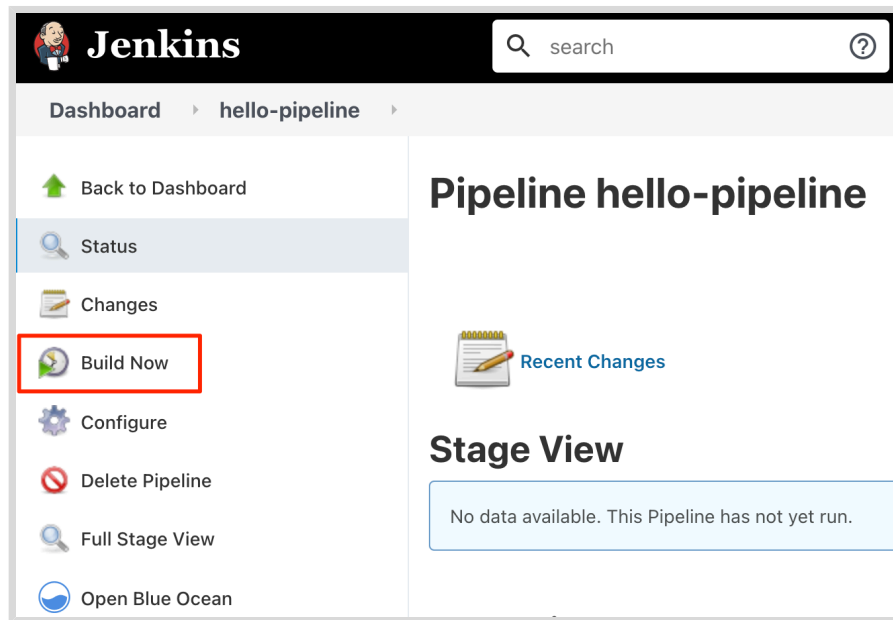
  stages {
    stage('Hello') {
      steps {
        echo 'Hello World'
      }
    }
  }
}
```

This defines a pipeline that runs by using any available Jenkins agent. It is a single-stage pipeline. The stage simply echoes "Hello World" to the output.

1.7. Click **Save** to save the pipeline and go to the pipeline details page.

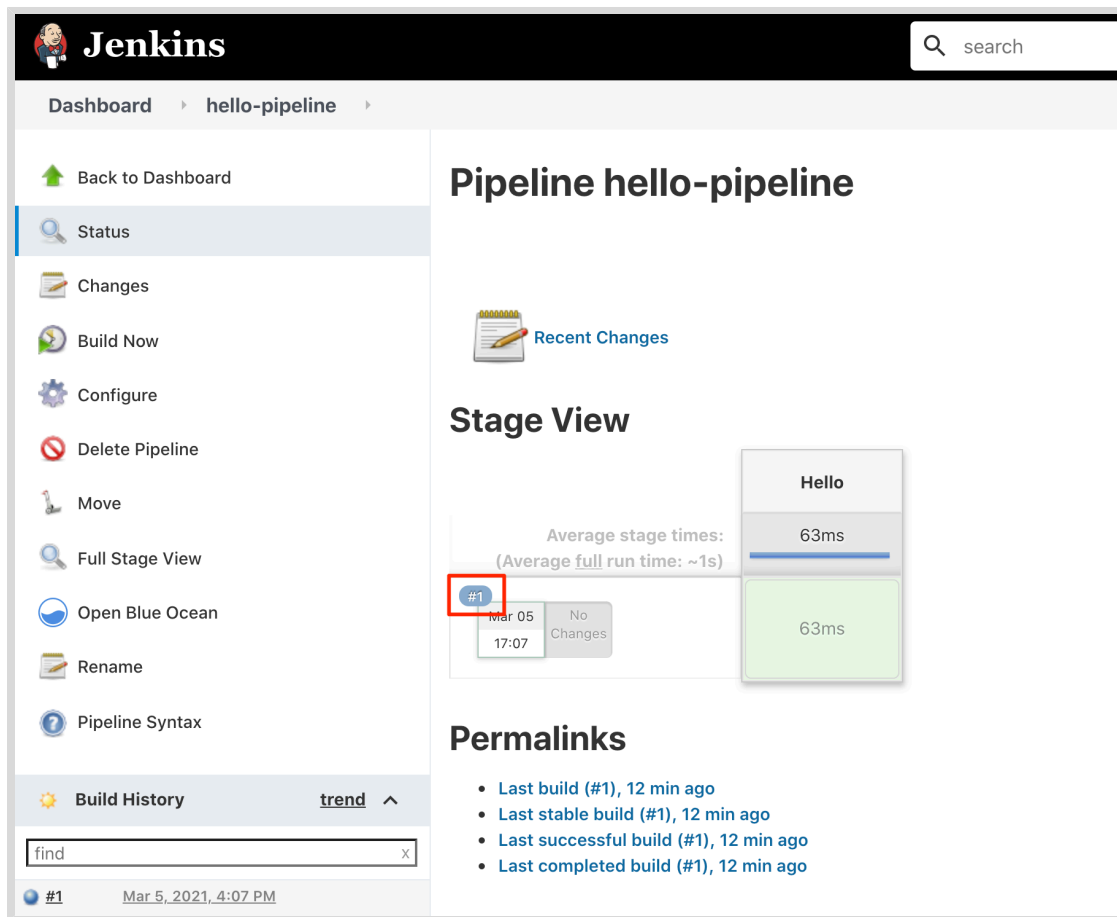
► **2.** Run the hello-pipeline job.

2.1. In the left navigation pane, click **Build Now** to schedule a pipeline run.

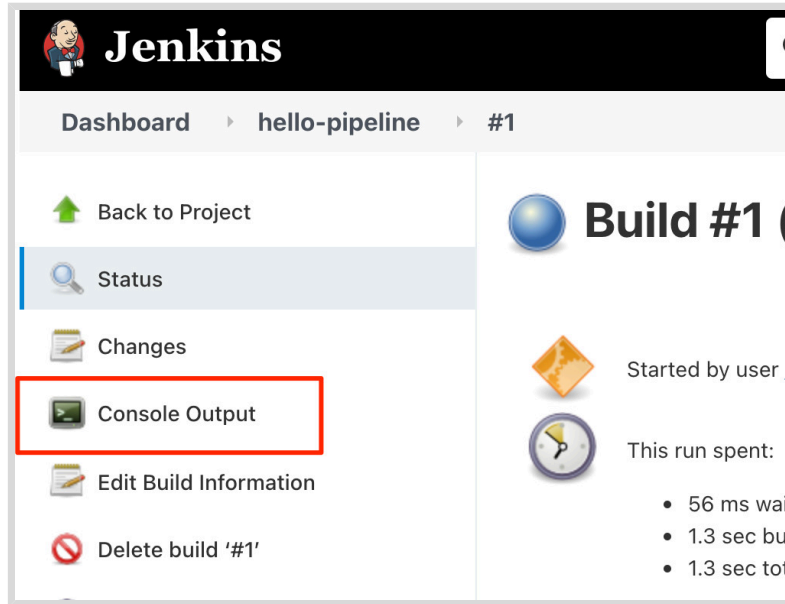


While it runs, Jenkins shows the progress of the different stages of the pipeline. At first, it might take several seconds for the job status to appear.

- 2.2. After the build has finished, click the newly added build number or build date to go to the build details page.



- 2.3. On the build page, click **Console Output** in the left navigation pane.

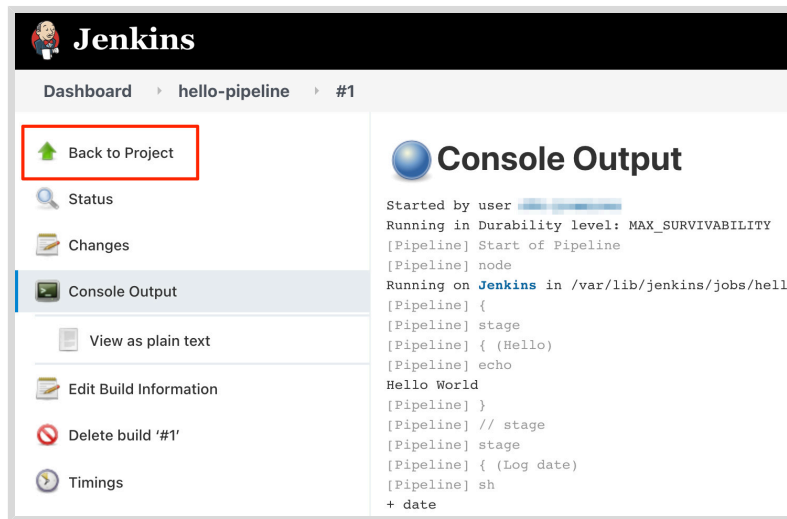


The **Console Output** page shows the output for the build. It includes any console output from the commands run during the execution of the pipeline.

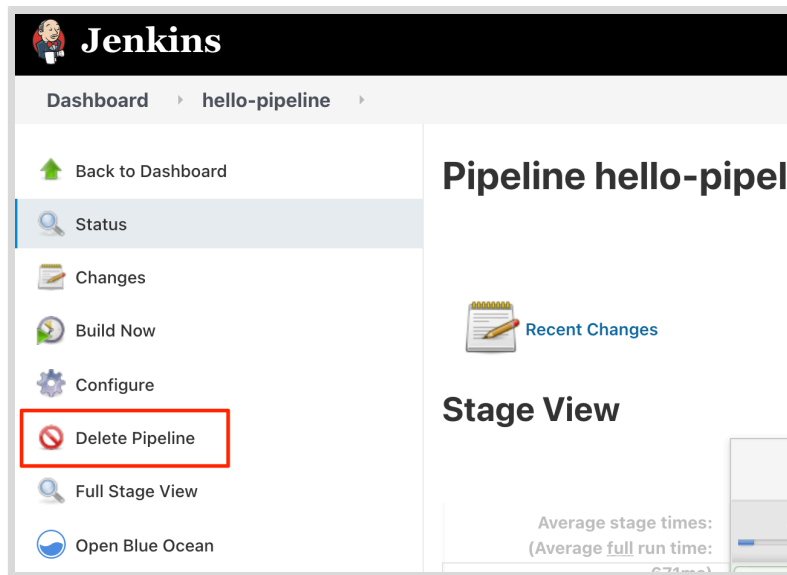
```
...output omitted...
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/jobs/hello-pipeline/workspace
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Hello)
[Pipeline] echo
Hello World
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Notice that the **Hello** stage echoed **Hello World**.

- 2.4. Click **Back to Project** in the left navigation pane.



- 2.5. Click **Delete Pipeline** in the left navigation pane. A dialog displays requesting confirmation. Click OK to delete the pipeline.



After deleting the pipeline, Jenkins redirects you to the main page.

- ▶ **3.** Create a pipeline with multiple steps.
 - 3.1. Click **New Item** in the left pane to create a new pipeline.
 - 3.2. Enter `multi-pipeline` as the item name. Click **Pipeline** and then click **OK**.
 - 3.3. Click the **Pipeline** tab and enter the following code in the **Script** field:

```
pipeline {
  agent any

  stages {
    stage('Hello') {
      steps {
```



```

        echo 'Hello World'
    }
}
stage('Log date') {
    steps {
        sh 'date'
    }
}
}
}

```

The preceding code defines a declarative pipeline with two stages. As in the previous pipeline, the first stage simply echoes `Hello World`. The second stage runs the `date` command from a shell. This echoes the current date and time.

Because the stages are independent, this job could be made to continue even when the `Hello` stage fails.

- 3.4. Click **Save** to save the pipeline and go to the pipeline details page.
- 3.5. In the left pane, click **Build Now** to schedule a pipeline run.
- 3.6. On the build page, click **Console Output** in the left pane to see the output from each stage.

```

...output omitted...
[Pipeline] stage
[Pipeline] { (Hello) ❶
[Pipeline] echo
Hello World
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Log date) ❷
[Pipeline] sh
+ date
<timestamp for when the build ran>
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS ❸

```

❶ ❷ Each stage is demarcated within the output.

❸ The final result of the job is either **SUCCESS** or **FAILURE**.

- 3.7. Click **Back to Project**, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.
- ▶ 4. Create a pipeline that runs a simple test command.
 - 4.1. Click **New Item** in the left pane to create a new pipeline.
 - 4.2. Enter `story-count-test` as the item name. Click **Pipeline** and then click **OK**.
 - 4.3. Click the **Pipeline** tab and enter the following code in the **Script** field:

```

pipeline {
  agent any

  stages {
    stage('Checkout') {
      steps {
        git branch: 'main', url: 'https://github.com/RedHatTraining/D0400-
apps'
      }
    }
    stage('Test Word Count') {
      steps {
        sh './story-count/test-wc.sh ./story-count/frankenstein.txt 500'
      }
    }
  }
}

```

The preceding declarative pipeline also has two stages. The **Checkout** stage uses Git to clone the repository and check out the **story-count** branch. The **Test Word Count** stage runs a test command which asserts a text file that contains the specified number of words.

Unlike the previous pipeline, the **Test Word Count** stage is dependent on the **Checkout** stage. If the first stage fails to clone the code, the second stage cannot work.

- 4.4. Click **Save** to save the pipeline and go to the pipeline details page.
- 4.5. In the left pane, click **Build Now** to schedule a pipeline run.
- 4.6. On the build page, click **Console Output** in the left pane to see the output from each stage.

```

[Pipeline] Start of Pipeline
...output omitted...
[Pipeline] stage
[Pipeline] { (Test Word Count)
[Pipeline] sh
+ ./story-count/test-wc.sh ./story-count/frankenstein.txt 500
FAIL: got 433 but expected 500
...output omitted...
[Pipeline] End of Pipeline
ERROR: script returned exit code 1
Finished: FAILURE

```

The build fails at the **Test Word Count** stage. The test script received 500 as the expected number of words for the document, but the actual number was 433.

- ▶ 5. Correct the shell command in the test stage.
 - 5.1. From the details page for the **story-count - test** job, click **Configure** in the left pane.

- 5.2. Click the **Pipeline** tab and update the code in the **Script** field. Correct the argument in the pipeline script by changing it from 500 to 433.

```
...output omitted...
stage('Test Word Count') {
  steps {
    sh './story-count/test-wc.sh ./story-count/frankenstein.txt 433'
  }
}
...output omitted...
```

- 5.3. Click **Save** to save the pipeline changes and go to the pipeline details page.
- 5.4. In the left pane, click **Build Now** to schedule a pipeline run.
- 5.5. On the build page, click **Console Output** in the left pane to see the output from each stage. The pipeline finishes successfully.

```
[Pipeline] Start of Pipeline
...output omitted...
[Pipeline] stage
[Pipeline] { (Checkout)
[Pipeline] git
...output omitted...
[Pipeline] stage
[Pipeline] { (Test Word Count)
[Pipeline] sh
+ ./story-count/test-wc.sh ./story-count/frankenstein.txt 433
SUCCESS: correct word count
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS
```

- 5.6. Click **Back to Project**, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.

This concludes the guided exercise.

Creating Scripted Pipelines

Objectives

After completing this section, you should be able to incorporate shell scripts into Jenkins pipelines.

Comparing Declarative and Scripted Pipelines

In *Building a Basic Declarative Pipeline* section we described the structure and usage of declarative pipelines. These kinds of pipelines are good for simple tasks but are limited by the rigidity of their structure.

Scripted pipelines offer an imperative programming style by using the Groovy programming language. Scripted pipelines do not have a strict structure and syntax to follow, and are only limited to constraints within Groovy.

Declarative and Scripted pipelines both implement forms of pipelines, but with slight differences.

Main differences between declarative and scripted pipelines

Declarative Pipelines	Scripted Pipelines
Follows a declarative programming model	Follows an imperative programming model
Simplified and opinionated syntax	Inherits the syntax from the Groovy language
Strict and predefined structure	Flexible structure
Ideal for continuous delivery pipelines	Ideal for advanced users with complex requirements

Writing a Scripted Pipeline

The versatility of scripted pipelines makes them suitable for complex tasks. But this same versatility can end up making your pipeline too complex and difficult to maintain. Here are some best practices to write more maintainable pipelines:

- Enclosing your pipeline definition within a `node` block. Jenkins will schedule the execution of the stages on a queue and create a dedicated workspace for the pipeline.

The following example illustrates the use of the `node` block, and the `stage` directive.

```
node {
    stage('Hello') {
        echo 'Hello Ravi!'
    }
}
```

- Encapsulate subsets of steps with a common objective within `stage` directives.
- Avoid complex steps by delegating execution to underlying frameworks.

For example, instead of creating a stage with multiple `sh` tasks, create a shell script aggregating all actions and use a single `sh` task invoking the script.

Just like declarative pipelines, you can only use steps that are compatible with the Pipeline plug-in. Some of the most used steps are `echo`, `git`, and `sh`.



Note

This section focuses on creating a basic scripted pipeline, without performing a deep dive into the Groovy language. You can find more information about the Groovy language in the reference links.

Controlling the Flow

Jenkins executes all the instructions defined in a scripted pipeline in order from top to bottom. To make this way of processing the instructions more flexible, you can use Groovy expressions to control the execution flow of your pipeline. With statements such as conditionals or loops, you can define the order of execution.

Variables

Variables are not a control flow mechanism, but you can use them to store values needed in the different stages of your pipeline. You can define variables by using either the `type`, or the `def` keyword followed by the variable name.

The following example illustrates the definition and use of variables.

```
String person = 'Zach' 1
def anotherPerson = 'Richard' 2
node {
    stage('Hello') {
        echo "Hello ${person} and ${anotherPerson}!" 3
    }
}
```

- ¹ Variable definition specifying the type
- ² Variable definition without specifying the type
- ³ Variable usage with string interpolation

Conditionals

Conditionals allow you to perform different actions depending on a boolean condition. For example, you can execute the build of your application only if the pipeline execution is in the `main` branch.

The following example shows the use of the standard `if/else` statement in a scripted pipeline.

```
node {
    stage('Hello') {
        if (env.BRANCH_NAME == 'main') {
            echo 'Hello Manuel!'
        } else {
```

```

        echo 'Hello Enol!'
    }
}

```

The preceding example echoes different strings depending on whether the pipeline is executed in the `main` branch or not.

You can use other conditional statements in your pipeline scripts, for example the `switch/case` statement. Switch statements work like a set of chained `if`-statements. For long conditional chains, switch statements are often easier to read.

Loops

A loop is a structure that repeats a block of instructions until a condition is met. For example, you can store the name of your testing suites in a variable, loop through all the values, and execute the testing suites one after another.

One of the most common loop statements is `for`. The following example illustrates the standard use of the `for` statement.

```

def people = ['Eduardo', 'Jordi']
node {
    stage('Hello') {
        for (int i = 0; i < people.size(); ++i) {
            echo "Hello ${people[i]}!"
        }
    }
}

```

The preceding example loops through the values of the `people` variable and echoes a message.

Other forms of loop statements include `do` and `do/while`.

Functions

Functions perform specific pieces of work by encapsulating a set of reusable instructions. When you find yourself repeating the same function in multiple pipelines, a good practice is to instead move the function to a shared library. That way, you reduce the maintenance cost of your pipelines and, at the same time, improve the readability of the pipeline scripts.

The following example illustrates the definition and usage of functions.

```

node {
    stage('Hello') {
        salute('Marek') ❶
    }
}

def salute(person) { ❷
    echo "Hello ${person}!"
}

```

^❶ Usage of the `salute` function

2 Definition of the `salute` function

Combining Declarative and Scripted Pipelines

Declarative pipelines have a strict and opinionated syntax. This simple syntax might not be sufficient for your project requirements. For example, you might need to loop over a range of values to perform certain actions in one of the stages of your pipeline.

To make your declarative pipelines more flexible and easy to adapt to your requirements, you can define scripted blocks inside your declarative pipelines. The `script` step wraps scripted pipeline blocks inside your declarative pipelines.

The following example illustrates the use of the `script` step in a declarative pipeline.

```
pipeline {
  agent any
  stages {
    stage('Hello') {
      steps {
        script {
          def people = ['Sam', 'Dave']
          for (int i = 0; i < people.size(); ++i) {
            echo "Hello ${people[i]}"
          }
        }
      }
    }
  }
}
```

The preceding declarative pipeline includes a scripted pipeline block, which defines a variable as a list of two values. The `for` statement loops through the values and prints a message with each.



References

The Apache Groovy programming language - Syntax

<https://groovy-lang.org/syntax.html>

Jenkins Pipeline Steps reference

<https://www.jenkins.io/doc/pipeline/steps/>

Pipeline examples

<https://github.com/jenkinsci/pipeline-examples>

Extending with Shared Libraries

<https://www.jenkins.io/doc/book/pipeline/shared-libraries/>

► Guided Exercise

Creating Scripted Pipelines

In this exercise you will learn how to use scripted pipelines.

Outcomes

You should be able to write scripted pipelines and differentiate them from declarative pipelines.

Before You Begin

To perform this exercise, ensure you have:

- Access to a Jenkins server
- A web browser such as Firefox or Chrome



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the `simple-webapp` application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps/simple-webapp` application folder and checkout the `main` branch of the `D0400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps/simple-webapp
[user@host simple-webapp]$ git checkout main
...output omitted...
```

- 1.2. Create a new branch to save any changes you make during this exercise and push it to your fork.

```
[user@host simple-webapp]$ git checkout -b scripted-pipelines
Switched to a new branch 'scripted-pipelines'
[user@host simple-webapp]$ git push -u origin scripted-pipelines
...output omitted...
* [new branch]      scripted-pipelines -> scripted-pipelines
Branch scripted-pipelines set up to track remote branch scripted-pipelines from origin.
```

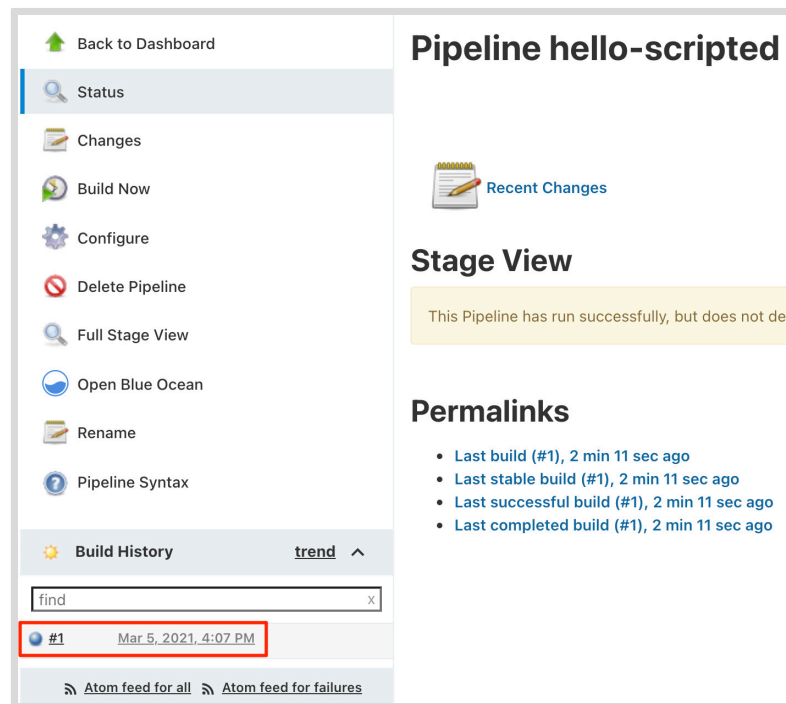
- 2. Create a basic scripted pipeline.

- 2.1. Open your web browser and navigate to the URL of the Jenkins server that you installed in the first guided exercise. Log in into Jenkins with your OpenShift credentials.
- 2.2. Click **New Item** in the left navigation pane.
- 2.3. Enter `hello-scripted` as the item name. Click **Pipeline** and then click **OK**.
- 2.4. Click the **Pipeline** tab and enter the following code in the **Script** field:

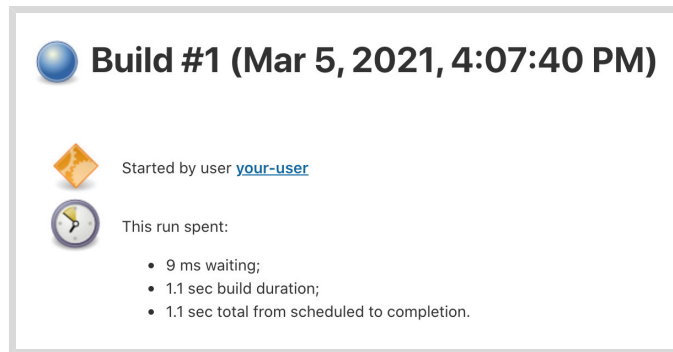
```
node {
  echo 'Hello world'
}
```

Scripted pipelines use the `node` keyword instead of the `pipeline` keyword used in declarative pipelines.

- 2.5. Click **Save** to save the pipeline and go to the pipeline details page.
- 2.6. In the left pane, click **Build Now** to schedule a pipeline run.
Jenkins shows the running pipeline in the **Build History** area located in the left pane. At first, it might take several seconds for the job status to appear.
- 2.7. After the build has finished, click the newly added build number or build date to go to the build details page.



- 2.8. Check that the build is successful. The blue ball icon indicates a successful build.



- 3. Create another scripted pipeline that runs the tests of a simple web application. The front-end includes a form to enter a name and show a greeting. The back-end receives the name and returns the greeting message to the front-end.

- 3.1. Click the **Jenkins** logo in the upper left to navigate to the Jenkins homepage.
- 3.2. Click **New Item** in the left pane.
- 3.3. Enter **webapp-tests** as the item name. Select the **Pipeline** option and click **OK**.
- 3.4. Click the **Pipeline** tab and enter the following code in the **Script** field:

```
node('nodejs') { ❶
  stage('Checkout') { ❷
    git url: 'https://github.com/YOUR_GITHUB_USER/DO400-apps', branch:
    'scripted-pipelines'
  }

  stage('Test') { ❸
    sh 'node ./simple-webapp/backend/test.js'
    sh 'node ./simple-webapp/frontend/test.js'
  }
}
```

Replace **YOUR_GITHUB_USER** with your GitHub user name to checkout your fork.

- ❶ The **nodejs** label indicates that the Node.js agent must execute this pipeline.
- ❷ ❸ This pipeline defines two stages to checkout the code and run the tests. Scripted pipelines do not need the **steps** block to define the steps in a stage.
- 3.5. Click **Save** to save the pipeline and go to the pipeline details page.
- 3.6. From the left side navigation, click **Build Now** to schedule a pipeline run. Wait for the build to finish.



Important

The Node.js agent pod might take time to start. If this happens, then the build will wait until the agent is ready.

- 3.7. After the build has finished, click the newly added build number or build date to go to the build details page. Check that the build is successful.

- 3.8. Click **Back to Project** in the left navigation pane to return to the pipeline details page.
- 4. Optimize the pipeline to only run the back-end test suite, the front-end test suite, or both, depending on the changes made in the last commit.
- 4.1. From the left side navigation of the pipeline details page, click **Configure** to edit the pipeline.
- 4.2. Implement a new function called `isChanged` to check whether a folder has been changed in the last commit. Add the function code after the `node` pipeline block. Next, modify the `Test` stage to do the following:
- Execute the back-end tests only when the last commit includes changes in the `backend` folder.
 - Execute the front-end tests only when the last commit includes changes in the `frontend` folder.

Click the **Pipeline** tab, and then modify the **Script** field. The pipeline should look as follows:

```
node('nodejs') {
  stage('Checkout') {
    git url: 'https://github.com/YOUR_GITHUB_USER/DO400-apps', branch:
'scripted-pipelines'
  }

  stage('Test') {
    if (isChanged('simple-webapp/backend')) {
      echo 'Detected Backend changes'
      sh 'node ./simple-webapp/backend/test.js'
    }
    if (isChanged('simple-webapp/frontend')) {
      echo 'Detected Frontend changes'
      sh 'node ./simple-webapp/frontend/test.js'
    }
  }
}

def isChanged(dir) {
  changedFilepaths = sh(
    script: 'git diff-tree --no-commit-id --name-only -r HEAD',
    returnStdout: true
  ).trim().split('\n')

  for (filepath in changedFilepaths) {
    if (filepath.startsWith(dir)) {
      return true;
    }
  }
  return false
}
```

**Note**

Usually you will check the changes made since the last successful build. The preceding pipeline only checks the latest commit for the sake of simplicity.

- 4.3. Click **Save** to save the pipeline and go to the pipeline details page. Leave this browser tab open.
- 4.4. Open your editor and refactor the `backend/server.js` file, which is located in the `D0400-apps/simple-webapp` folder. Modify the code to extract the `greet` call to a constant.

```
const server = http.createServer((req, res) => {
  const { name } = url.parse(req.url, true).query;
  const greeting = greet(name);

  res.statusCode = 200;
  res.setHeader("Content-Type", "text/plain");
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.end(greeting);
});
```

Save the file.

- 4.5. Verify that unit tests pass after refactoring the code. Check that the exit code is zero, which means that the tests have passed.

```
[user@host simple-webapp]$ node backend/test.js
[user@host simple-webapp]$ echo $?
0
```

**Important**

Windows users might have a different output in PowerShell:

```
PS C:\Users\user\D0400\D0400-apps\simple-webapp> echo $?
True
```

- 4.6. Commit and push the changes.

```
[user@host simple-webapp]$ git commit -a -m "Refactor"
[user@host simple-webapp]$ git push origin
```

- 4.7. Go back to the pipeline details page in the web browser. From the left side navigation, click **Build Now** to schedule a pipeline run.
- 4.8. When the new build appears in the **Build History** pane, click the new build number to go to the build page.
- 4.9. In the left navigation, click **Console Output** and verify that the pipeline only executes the back-end tests.

```
...output omitted...
Detected Backend changes
[Pipeline] sh
+ node ./simple-webapp/backend/test.js
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

- 4.10. Click **Back to Project**, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.
- ▶ 5. Use a script block in a declarative pipeline to combine the declarative and the scripted approaches. The pipeline runs a test that checks the HTTP response code of the back-end application.
 - 5.1. Click **New Item** in the left pane to create a new pipeline.
 - 5.2. Enter **webapp-api-test** as the item name. Select the **Pipeline** option and click **OK**.
 - 5.3. Click the **Pipeline** tab and enter the following code in the **Script** field:

```
pipeline {
    agent {
        node {
            label 'nodejs'
        }
    }
    stages {
        stage('Checkout') {
            steps {
                git url: 'https://github.com/YOUR_GITHUB_USER/D0400-apps', branch:
'scripted-pipelines'
            }
        }
        stage('Test') {
            steps {
                sh 'simple-webapp/backend/test_api.sh'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying...'
            }
        }
    }
}
```

Replace **YOUR_GITHUB_USER** with your GitHub user name to checkout your fork.

This pipeline checks out the code, executes an API test and runs a deployment stage. The API test runs the server, makes a request, and verifies that the HTTP response code is 200.

- 5.4. Click **Save**.
- 5.5. In the pipeline details page, Click **Build Now**.
- 5.6. Navigate to the build output page and verify that the pipeline fails.

```
...output omitted...
+ simple-webapp/backend/test_api.sh
curl: no URL specified!
...output omitted...
[Pipeline] { (Deploy)
Stage "Deploy" skipped due to earlier failure(s)
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
ERROR: script returned exit code 1
Finished: FAILURE
```

The pipeline fails because the `test_api.sh` requires an HTTP verb parameter. The failure also causes the pipeline to skip the **Deploy** stage.

- 5.7. Open the pipeline configuration page and fix the test stage. Go back to the pipeline details page and click **Configure** in the left navigation page.
Click the **Pipeline** tab and in the **Script** field. Introduce a `script` block in the **Test** stage to run the `test_api.sh` script with different HTTP verbs. Run the test with the **GET**, **POST** and **PUT** verbs.

```
stage('Test') {
  steps {
    script {
      def verbs = ['GET', 'POST', 'PUT']
      for (int i = 0; i < verbs.size(); ++i) {
        sh "simple-webapp/backend/test_api.sh ${verbs[i]}"
      }
    }
  }
}
```

The `script` block allows the addition of complex scripted logic in declarative pipelines.



Important

Pay attention to the double quotes in the `test_api.sh` script call. Double quotes are necessary to interpolate the `${verbs[i]}` expression.

- 5.8. Click **Save** to save the changes and go back to the pipeline details page.

- 5.9. Click **Build Now** to run the pipeline. In the **Stage View**, observe the build progress and check that the pipeline passes.
- 5.10. Click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.

This concludes the guided exercise.

Controlling Step Execution

Objectives

After completing this section, you should be able to create checkpoints, parallel executions, and conditionals for more advanced pipeline behavior.

Creating Pipelines as Code

Pipeline as code is a technique that encourages developers to treat the pipeline configuration as code. As with regular software code, you must track your pipeline configuration by using a version control system.

When you follow the pipeline as code technique, you usually keep the pipeline configuration in the same repository as the application code. The main advantages of this technique are the following:

- All the changes to the pipeline are trackable and auditable.
- The pipeline configuration is open and accessible to developers.

In Jenkins, you can follow this technique and store the pipeline script in a file, usually named **Jenkinsfile**. In this file you can use the declarative, or the scripted syntax to describe your pipeline.

```
pipeline {
  agent { node { label 'nodejs' } }
  stages {
    stage('Install Dependencies') {
      steps {
        sh 'npm install'
      }
    }
    stage('Test') {
      steps {
        sh 'npm test'
      }
    }
  }
}
```

The preceding example displays the content of a **Jenkinsfile** file. The pipeline described in the example uses a declarative syntax, and it defines the two stages involved in this pipeline. The first stage installs the dependencies and the second runs the tests.

Integrating Jenkins and GitHub

In Jenkins, a *project* is a way to group the definition and configuration of automated tasks. You can store the pipeline definition in a version control system and configure Jenkins to gather it.

When configuring a project, navigate to the **Pipeline** area. From the **Definition** list, select **Pipeline script from SCM** from **SCM**. Select **Git** from the **SCM** list, and fill the Git repository information.

The **Branch Specifier** field indicates in which Git branches Jenkins will execute the pipeline. This field accepts expressions to match multiple branches. For example, the **feature-*** expression indicates that Jenkins can execute the pipeline in all the branches that have **feature-** as a prefix.

Figure 5.14: Pipeline as code in Jenkins

The **Script Path** field indicates the location and name of the file that contains the pipeline definition. Usually, the file that contains the pipeline definition is named **Jenkinsfile**, and it is stored in the root of the repository.

When a project is associated with a repository, Jenkins uses this repository as the source of truth for the pipeline definition. First Jenkins pulls all the code from the repository, and then processes the pipeline definition.



Note

Having the pipeline definition, and the application code in the same repository has a major benefit. The implicit checkout stage in declarative pipelines will pull your application code, and the pipeline definition at the same time. Thanks to that, you do not need to create a stage to pull your application code.

Controlling Step Execution

The different directives included in Jenkins allow you to adapt the pipeline to the requirements of your project.

Supplying Pipeline Parameters

Jenkins allows you to parameterize your pipelines. This feature helps you to reuse your pipelines, or build more complex flows. A pipeline can declare parameters with values passed to the underlying tasks, parameters that decide the execution of some steps, or that restrict the agents that can execute the pipeline. For example, by using a pipeline parameter, you can control when to

deploy a release to a particular environment, such as production. Or, in case of a rollback, you can use a pipeline parameter to specify the commit to restore.

With the `parameters` directive you can define a list of parameters to use in the execution of the pipeline. In this list, you must define at least the type and a name for the parameter. Optionally, you can define a default value or a description.

The declarative syntax supports different types of parameters.

- `string`: to store strings.
- `text`: to store multiline strings.
- `booleanParam`: to store boolean values.
- `choice`: to store the value selected from a list of options.
- `password`: to stores credentials.

You can access the pipeline parameters, and their assigned values by using the `params` object. The following example describes a declarative pipeline that uses a string parameter:

```
pipeline {
  agent any
  parameters { 1
    string( 2
      name: 'PERSON', 3
      defaultValue: 'John' 4
    )
  }
  stages {
    stage('Echo') {
      steps {
        echo "Hello ${params.PERSON}!" 5
      }
    }
  }
}
```

- 1 Initializes the definition of the pipeline parameters.
- 2 Defines the parameter type.
- 3 Defines the parameter name.
- 4 Sets the default value.
- 5 Uses the `params` object to access to the value of the `PERSON` parameter, and prints it.



Note

In declarative pipelines, you can use the `parameters` directive only once inside the pipeline block.

Using Checkpoints

In some cases, you might need to request confirmation from a user to execute a stage. For example, you might request confirmation before dropping a database or replacing a production application instance. Essentially, it is a good idea to ask for confirmation before any destructive task.

In this case, use the `input` directive in your pipeline scripts. This instructs Jenkins to pause the stage during execution and prompt for input. When you approve the input, the stage execution continues, otherwise it aborts the execution.

You must define at least the `message` option. Other configuration options are:

- `id`: sets an optional identifier for the input.
- `submitter`: defines a list of users allowed to submit the input.
- `ok`: configures an alternative text for the approve button.
- `parameters`: defines a list of parameters to prompt to the user to provide input. Those parameters are available in the environment only for the execution of the stage.

The following example describes a declarative pipeline that requests confirmation before the **Echo** stage is executed:

```
pipeline {
  agent any
  stages {
    stage('Echo') {
      input { ❶
        message "Do you want to salute?" ❷
      }
      steps {
        echo "Hello!"
      }
    }
  }
}
```

The preceding pipeline script defines:

- ❶ The input directive.
- ❷ The message to display to the user when the execution is paused.

Using Conditionals

In more advanced pipelines, you might need to run a stage if certain conditions are met. It is common that some stages only run on selected code branches or environments. For example, executions on development branches avoid executing slow functional tests.

In this case, you can use the `when` directive in your pipeline scripts. The `when` directive determines if a stage is executed depending on a given condition. You can build nested and complex conditions with the `not`, `allOf`, or `anyOf` built-in conditions.

```

pipeline {
    agent any
    stages {
        stage('Echo') {
            when { ❶
                expression { env.GIT_BRANCH == 'origin/main' } ❷
            }
            steps {
                echo 'Hello from the main branch!'
            }
        }
    }
}

```

The preceding pipeline script defines:

- ❶ The **when** directive.
- ❷ The expression that determines when the stage is executed. In this case, the **Echo** stage is executed only in the **origin/main** branch.

The expression built-in condition evaluates instructions from the Groovy programming language. It only executes the stage if the result of evaluating the instruction is **true**. Keep in mind that **strings** evaluate to **true** and **null** evaluates to **false**.



Note

The Groovy **match** operator (**==~**) allows to match a string against a regular expression. For example, **"branch-test" ==~ /. *test/** will be matched as **branch-test** is a string ending by **test**.

By default, the **input** directive takes precedence over the evaluation of the **when** conditions. You can add **beforeInput true** within the **when** block to change this behavior.

Defining the Pipeline Execution

Pipelines define a series of stages executed in a specific order. Jenkins supports three different types of stage executions: **sequential**, **parallel**, and **matrix**. In the same pipeline you can mix the three types of executions to adapt the pipeline to your needs.



Important

A stage must have only one of the following directives: **steps**, **stages**, **parallel**, or **matrix**.

Sequential Execution

In this type of execution, Jenkins executes all the stages included in the **stages** section in sequential order. This is the default execution mode in Jenkins.

The following diagram shows a pipeline where all the stages are sequential.

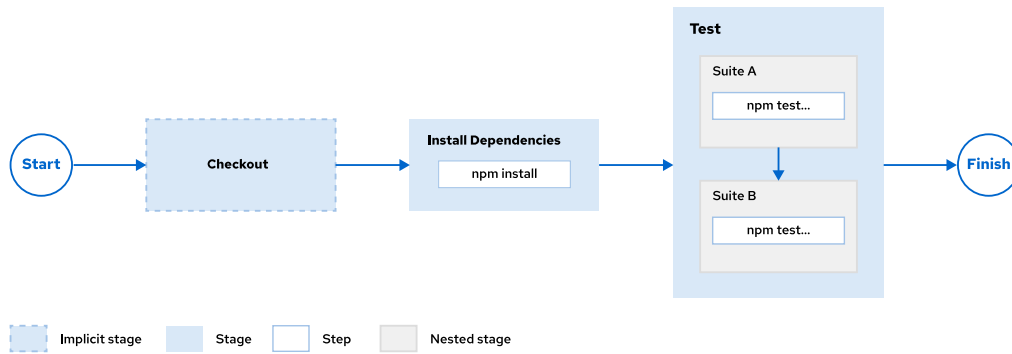


Figure 5.15: Example of sequential execution

In the preceding example, the execution order of the stages is:

1. The **Checkout** stage.
2. The **Install Dependencies** stage.
3. The **Test** stage, executing the nested stages in sequential order.
 - The stage named **Suite A**.
 - The stage named **Suite B**.

You can transform the preceding flow into code by using the declarative syntax. The pipeline script for the preceding pipeline is the following:

```
pipeline {
  agent { node { label 'nodejs' } }
  stages { ❶
    stage('Install Dependencies') {
      steps { sh 'npm install' }
    }
    stage('Test') {
      stages { ❷
        stage('Suite A') {
          steps { sh 'npm test -- -f testSuiteA' }
        }
        stage('Suite B') {
          steps { sh 'npm test -- -f testSuiteB' }
        }
      }
    }
  }
}
```

The preceding pipeline script defines:

- ❶ A list of stages to execute in sequential order.
- ❷ A list of nested stages to execute in sequential order.

Parallel Execution

In this type of execution, Jenkins executes all the stages included in the `parallel` section in parallel. Parallel execution of steps not only accelerates the execution of the pipeline but also enables a fail-fast approach on test-related steps.

The following diagram shows a pipeline that combines sequential and parallel stages.

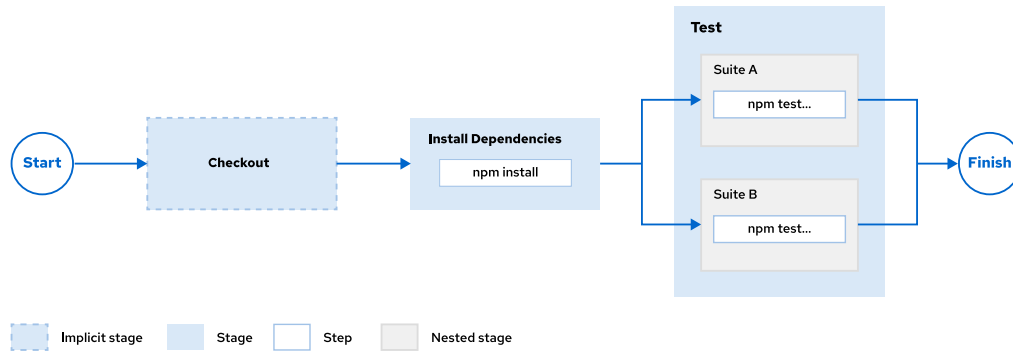


Figure 5.16: Example of parallel execution

In the preceding example, the execution order of the stages is:

1. The **Checkout** stage.
2. The **Install Dependencies** stage.
3. The **Test** stage, executing the nested stages in parallel.

You can transform the preceding flow into code by using the declarative syntax. The pipeline script for the preceding pipeline is the following:

```

pipeline {
  agent { node { label 'nodejs' } }
  stages { ❶
    stage('Install Dependencies') {
      steps { sh 'npm install' }
    }
    stage('Test') {
      parallel { ❷
        stage('Suite A') {
          steps { sh 'npm test -- -f testSuiteA' }
        }
        stage('Suite B') {
          steps { sh 'npm test -- -f testSuiteB' }
        }
      }
    }
  }
}

```

The preceding pipeline script defines:

- ❶ A list of stages to execute in sequential order.

- 2 A list of nested stages to execute in parallel order.

By default, Jenkins continues the execution of all the parallelized stages even if one of them fails. You can change this behavior by adding the `failFast true` option to the stage definition.

Matrix Execution

Sometimes a pipeline needs to execute the same steps multiple times with different combinations of parameters. Instead of declaring sequential or parallel stages for each combination, Jenkins can create the combinations by using matrix execution.

In this type of execution, Jenkins uses a multidimensional matrix to create combinations of execution parameters. Jenkins uses each combination of parameters to run the stages of the pipeline in parallel.

This is one of the most complex types of executions in Jenkins.

In a declarative pipeline, the `matrix` section defines the multidimensional matrix of combinations and the stages to execute in each one of the combinations. The `matrix` section contains the following sections:

- A section named `axes`. This section sets the dimensions of the matrix by using a subsection named `axis`.
- A section named `stages`. This section defines the stages to execute in each one of the combinations.
- An optional section named `excludes`. This section defines filter expressions that indicate combinations to remove from the matrix.

The following diagram shows an example of a pipeline, which includes a matrix execution in the **Test** stage. The combinations include all the values from the **Platform** and **Architecture** dimensions. The values for the **Platform** dimension are: *RHEL*, and *Fedora*. The values for the **Architecture** dimension are: *32-bit*, and *64-bit*.

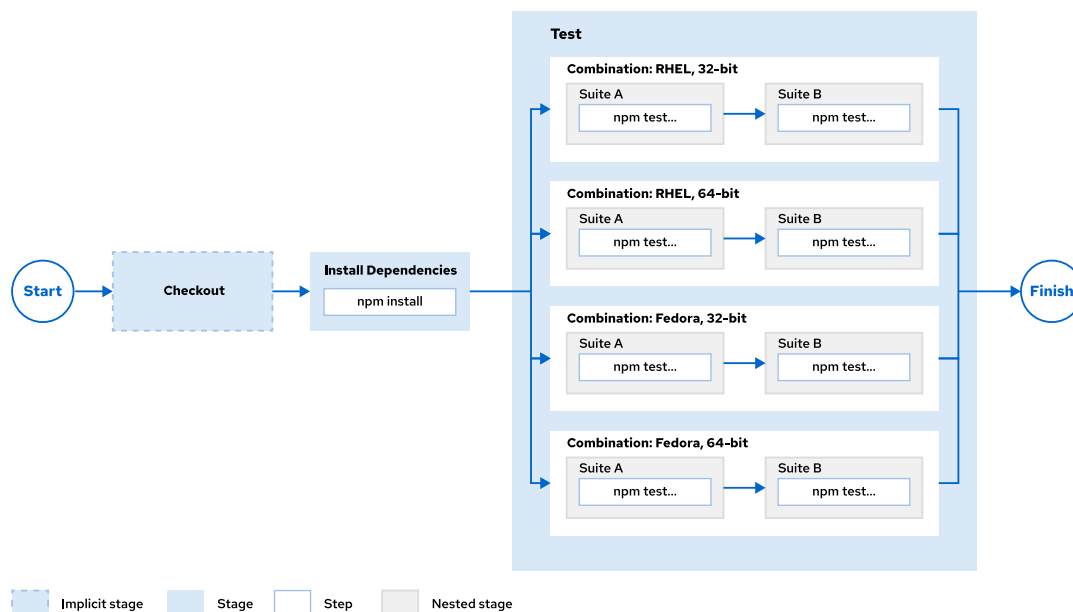


Figure 5.17: Example of matrix execution

In the preceding example, the execution order of the stages is:

1. The **Checkout** stage.
2. The **Install Dependencies** stage.
3. All the possible combinations of the matrix dimensions defined in the **Test** stage in parallel.
 - The stages **Suite A** and **Suite B** in sequence for the combination: *RHEL*, and *32-bit*.
 - The stages **Suite A** and **Suite B** in sequence for the combination: *RHEL*, and *64-bit*.
 - The stages **Suite A** and **Suite B** in sequence for the combination: *Fedora*, and *32-bit*.
 - The stages **Suite A** and **Suite B** in sequence for the combination: *Fedora*, and *64-bit*.

You can transform the preceding flow into code by using the declarative syntax. The script for the preceding pipeline is the following:

```
pipeline {
  agent none
  stages { ❶
    stage('Install Dependencies') {
      steps { sh 'npm install' }
    }
    stage('Test') {
      matrix { ❷
        axes { ❸
          axis { ❹
            name 'PLATFORM'
            values 'RHEL', 'Fedora'
          }
          axis { ❺
            name 'ARCHITECTURE'
            values '32-bit', '64-bit'
          }
        }
      }
      agent { ❻
        label "${PLATFORM}-${ARCHITECTURE}"
      }
      stages { ❼
        stage('Suite A') {
          steps { sh 'npm test -- -f testSuiteA' }
        }
        stage('Suite B') {
          steps { sh 'npm test -- -f testSuiteB' }
        }
      }
    }
  }
}
```

The preceding pipeline script defines:

- ❶ A list of stages to execute in sequential order.

- 2 The multidimensional matrix.
- 3 The dimensions of the matrix.
- 4 A dimension named **PLATFORM** with the values: RHEL, and Fedora.
- 5 A dimension named **ARCHITECTURE** with the values: 32-bit, and 64-bit.
- 6 The values of the combination define the agent executing the stages.
- 7 A list of stages to execute in each one of the combinations. The execution of the stages occurs in sequence.



References

Pipeline Syntax

<https://www.jenkins.io/doc/book/pipeline/syntax>

Welcome to the Matrix

<https://www.jenkins.io/blog/2019/11/22/welcome-to-the-matrix>

Groovy Operators

<https://groovy-lang.org/operators.html>

► Guided Exercise

Controlling Step Execution

In this exercise you will create a scripted pipeline with parallel stages, integrate the pipeline with GitHub, execute stages depending on the value of variables, and interact with the build.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to create a `Jenkinsfile`, integrate the pipeline with GitHub, parallelize tasks, use conditional steps, and include checkpoints.

Before You Begin

Ensure you have:

- Git installed
- A GitHub account
- Access to a Jenkins instance
- The `DO400-apps` repository cloned in your workspace folder
- A web browser such as Firefox or Chrome
- Access to a command line terminal



Important

Try to keep your course work organized by creating a workspace folder, such as `~/DO400`. Start the guided exercise from this folder.

Instructions

- 1. Create a public GitHub repository to store the source code of the sample application.
 - 1.1. Open a web browser and navigate to `https://github.com`.
 - 1.2. Sign in with your GitHub credentials. If you do not have an account, then create one.
 - 1.3. Click + → **New repository** in the upper right of the window.
 - 1.4. Fill in the **Repository name** field with the `do400-pipelines-control` value. Do not change any other field. Click **Create repository**.

- 1.5. Open a command line terminal and copy the contents of `~/D0400/D0400-apps/simple-webapp` to the `~/D0400/pipelines-control` folder. Navigate to the new directory.

```
[user@host D0400]$ cp -Rv ~/D0400/D0400-apps/simple-webapp/ \
~/D0400/pipelines-control
...output omitted...
[user@host D0400]$ cd ~/D0400/pipelines-control
[user@host pipelines-control]$
```



Important

Windows users must replace the preceding `cp` command in PowerShell as follows:

```
PS C:\Users\user\D0400> Copy-Item -Path D0400-apps\simple-webapp `
>> -Destination ~\D0400\pipelines-control -Recurse
```

- 1.6. Initialize a new local repository, commit all the application code, set `main` as the default branch, and add the remote repository.

```
[user@host pipelines-control]$ git init
Initialized empty Git repository in ...output omitted.../pipelines-control/.git/
[user@host pipelines-control]$ git add .
[user@host pipelines-control]$ git commit -m "Initial commit"
...output omitted...
create mode 100644 frontend/main.js
create mode 100644 frontend/test.js
[user@host pipelines-control]$ git branch -M main
[user@host pipelines-control]$ git remote add \
origin https://github.com/YOUR_GITHUB_USER/do400-pipelines-control.git
```

2. Create a scripted pipeline composed of different stages executed sequentially. The stages for the pipeline are the following:

- Checkout the repository.
- Run the backend tests.
- Run the frontend tests.

- 2.1. Create a file named `Jenkinsfile` in the root of the application folder to store the pipeline. The file contents should look like:

```
node('nodejs') {
    stage('Checkout') {
        git branch: 'main',
            url: 'https://github.com/YOUR_GITHUB_USER/do400-pipelines-control'
    }
    stage('Backend Tests') {
        sh 'node ./backend/test.js'
    }
    stage('Frontend Tests') {
```

```

    sh 'node ./frontend/test.js'
  }
}

```

In Jenkins you can store the pipeline in the Jenkins server by using the web console, or in code in a repository. Notice that the preceding code specifies the branch name.

**Note**

A good practice is to store the Jenkins pipeline in the repository along with the application code.

- 2.2. Stage and commit the `Jenkinsfile` file to the local repository.

```

[user@host pipelines-control]$ git add Jenkinsfile
[user@host pipelines-control]$ git commit -m "Added Jenkins integration"
[main eb6ad6c] Added Jenkins integration
1 file changed, 12 insertions(+)
create mode 100644 Jenkinsfile

```

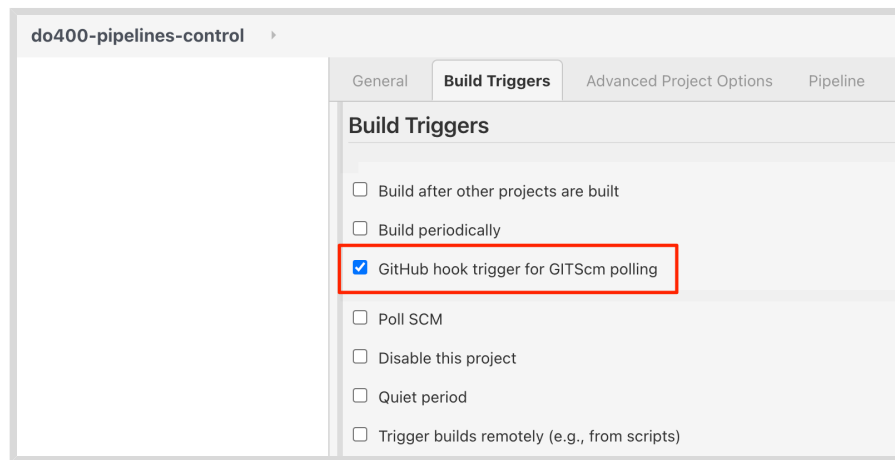
- 2.3. Push the local commits to the remote repository with the `git push` command. If prompted, enter your GitHub user name and personal access token.

```

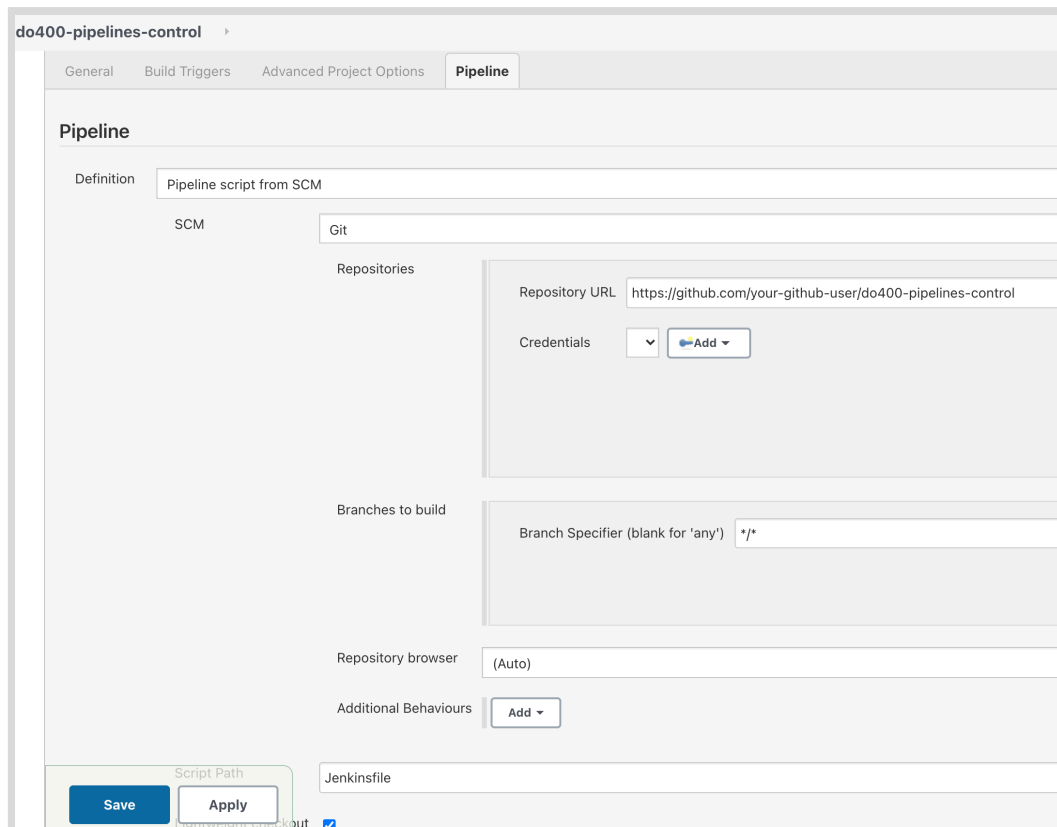
[user@host pipelines-control]$ git push origin HEAD
...output omitted...
Writing objects: 100% (13/13), 2.18 KiB | 2.18 MiB/s, done.
Total 13 (delta 0), reused 0 (delta 0)
To https://github.com/your_github_user/do400-pipelines-control.git
* [new branch]      HEAD -> main

```

- ▶ 3. Create a Jenkins project to run the pipeline defined in the `do400-pipelines-control` repository.
 - 3.1. Open a new tab in the browser and navigate to the URL of the Jenkins server that you installed in the first guided exercise.
 - 3.2. The Jenkins server is bound to your OpenShift account. If prompted, log in into Jenkins with your OpenShift credentials.
 - 3.3. In the left pane click **New Item**.
 - 3.4. Assign the name `do400-pipelines-control` to the project, then click **Pipeline**, and finally click **OK**.
 - 3.5. Click the **Build Triggers** tab, and then select the **GitHub hook trigger for GITScm polling** check box.



- 3.6. Click the **Pipeline** tab. In the **Pipeline** area, from the **Definition** list, select **Pipeline script from SCM**.
- 3.7. From the **SCM** list select **Git**. In the **Repository URL** type the URL of the GitHub repository created in the first steps.
The URL of the repository follows the pattern `https://github.com/YOUR_GITHUB_USER/do400-pipelines-control`.
- 3.8. In the **Branch Specifier** field, type `*/*`. The `*/*` value indicates that the pipeline will run on the detection of changes in any branch of the repository.



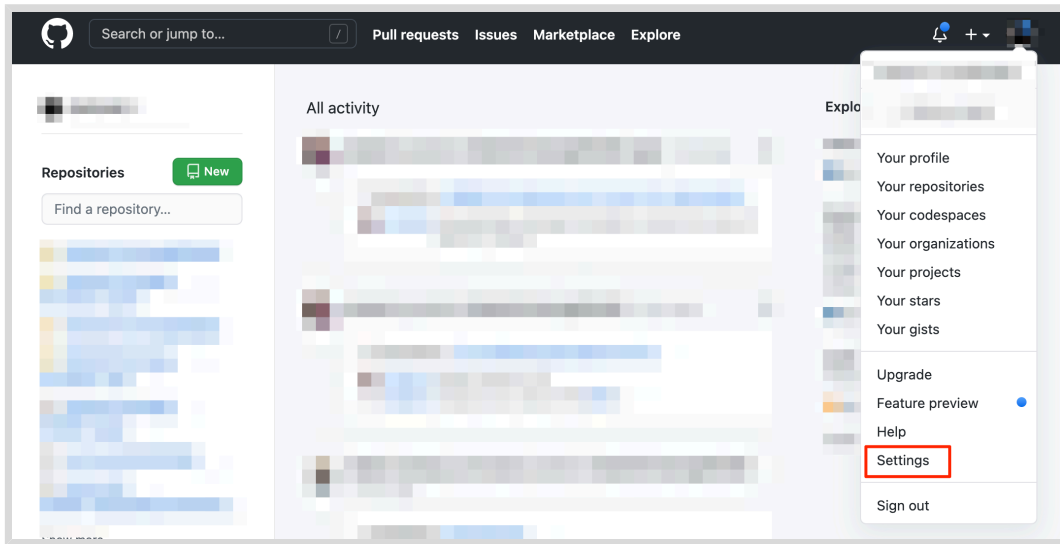
- 3.9. Click **Save** to save the pipeline and go to the pipeline details page.

- 3.10. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful.

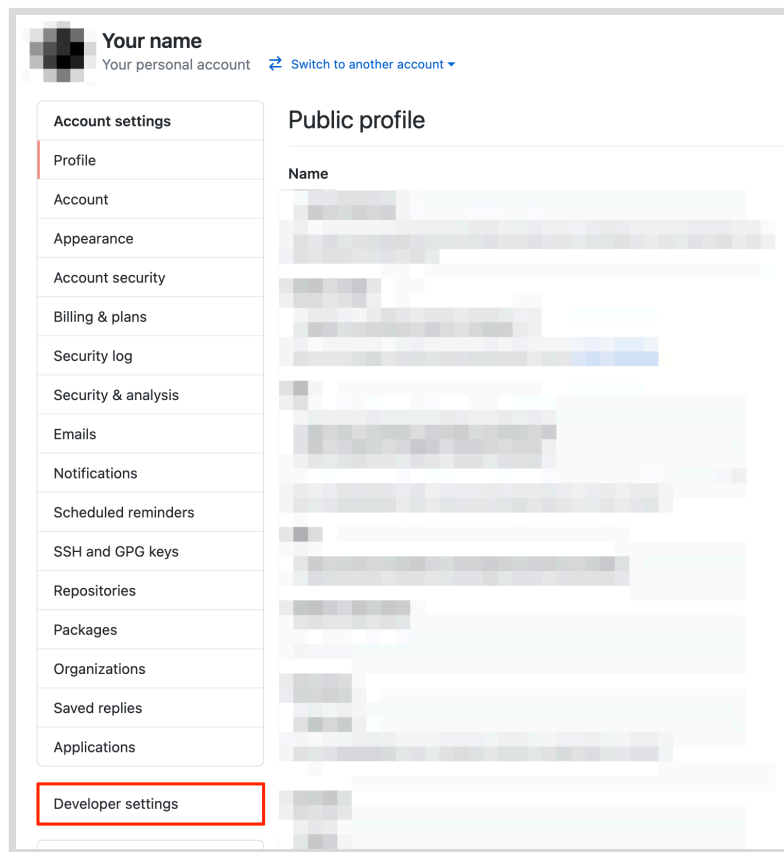
Jenkins first gathers the `Jenkinsfile` file stored in the GitHub repository, and then runs the pipeline defined in that file. While it runs, Jenkins shows the progress of the different stages of the pipeline in the **Stage View**. The background color of the different stages indicate the result of actions executed in the stage. A green color indicates that the stage was executed without problems and a red one indicates that there was an issue running that stage.

- 4. Integrate Jenkins with GitHub to run the pipeline on new changes pushed to the GitHub repository.

- 4.1. Return to the GitHub tab of your browser to generate an access token. Click your profile picture in the top right and then click **Settings**.



- 4.2. Click **Developer settings** in the left navigation pane.



- 4.3. In the developer settings page, click **Personal access tokens**, then click the **Generate new token** button in the right pane.
- 4.4. In the **Note** field, type `Jenkins Integration`. Select the `admin:repo_hook` check box, and then click **Generate token**.

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

Jenkins Integration

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

☐ repo	Full control of private repositories
☐ repo:status	Access commit status
☐ repo_deployment	Access deployment status
☐ public_repo	Access public repositories
☐ repo:invite	Access repository invitations
☐ security_events	Read and write security events
☐ workflow	Update GitHub Action workflows
☐ write:packages	Upload packages to GitHub Package Registry
☐ read:packages	Download packages from GitHub Package Registry
☐ delete:packages	Delete packages from GitHub Package Registry
☐ admin:org	Full control of orgs and teams, read and write org projects
☐ write:org	Read and write org and team membership, read and write org projects
☐ read:org	Read org and team membership, read org projects
☐ admin:public_key	Full control of user public keys
☐ write:public_key	Write user public keys
☐ read:public_key	Read user public keys
☒ admin:repo_hook	Full control of repository hooks
☒ write:repo_hook	Write repository hooks
☒ read:repo_hook	Read repository hooks
☐ admin:org_hook	Full control of organization hooks
☐ gist	Create gists

The GitHub page shows the generated token.

- 4.5. Copy the personal token for later use.



Warning

Make sure to copy the personal token because it is only displayed once and is needed in following steps.

- 4.6. Return to the Jenkins tab in your browser, and then click the **Jenkins** logo in the upper left to navigate to the Jenkins homepage.
- 4.7. Click **Manage Jenkins** in the left pane, scroll down to the **System Configuration** section, and then click **Configure System** in the right pane.

System Configuration

Configure System
Configure global settings and paths.

Global Tool Configuration
Configure tools, their locations and automatic installers.

Manage Plugins
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
▲ There are updates available

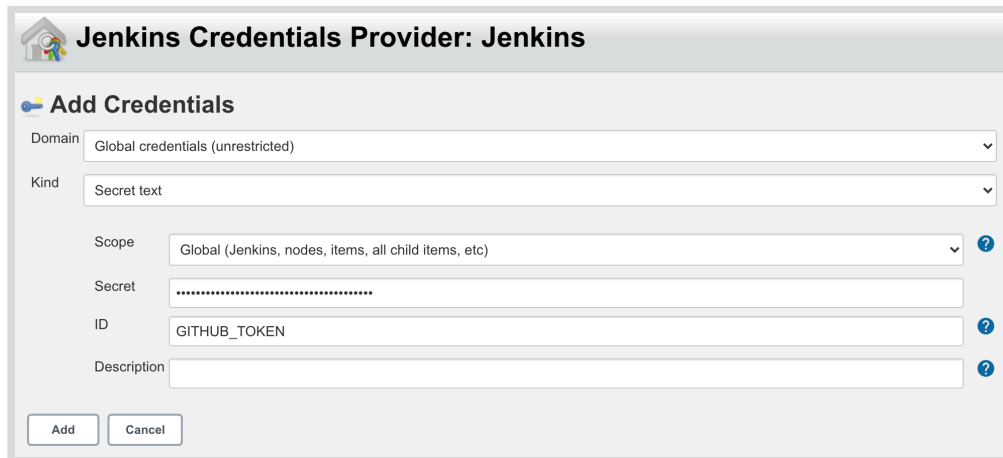
Manage Nodes and Clouds
Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

Configuration as Code
Reload your configuration or update configuration source

- 4.8. Scroll down to the **GitHub** area to add a GitHub server.
- 4.9. Click the **Add Github Server** list, then click **GitHub Server**.

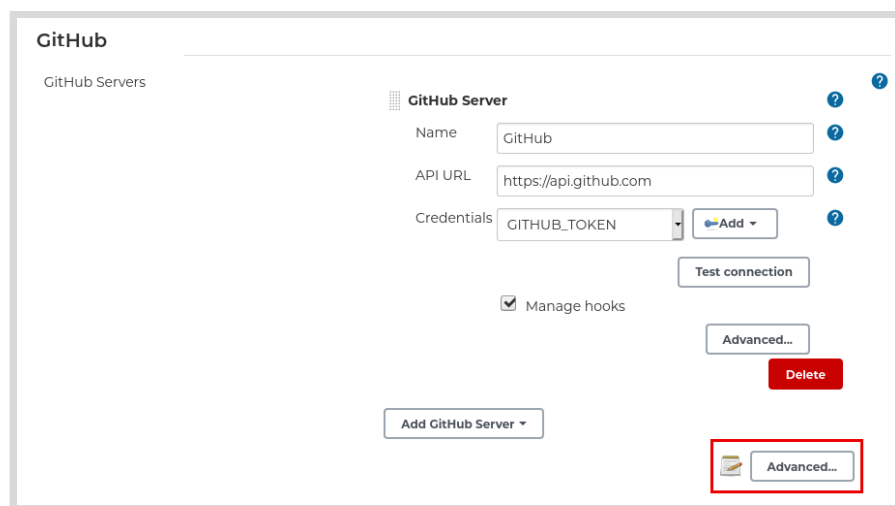


- 4.10. Type **GitHub** in the **Name** field. In the **Credentials** list select **Add**, and then **Jenkins** to open a window, which allows you to store credentials.
- 4.11. In the **Jenkins Credentials Provider: Jenkins** window, select **Secret** text from the **Kind** list. In the **Secret** field, paste or type the personal token generated in GitHub. Type **GITHUB_TOKEN** in the **ID** field.



Click **Add** to store the secret in Jenkins.

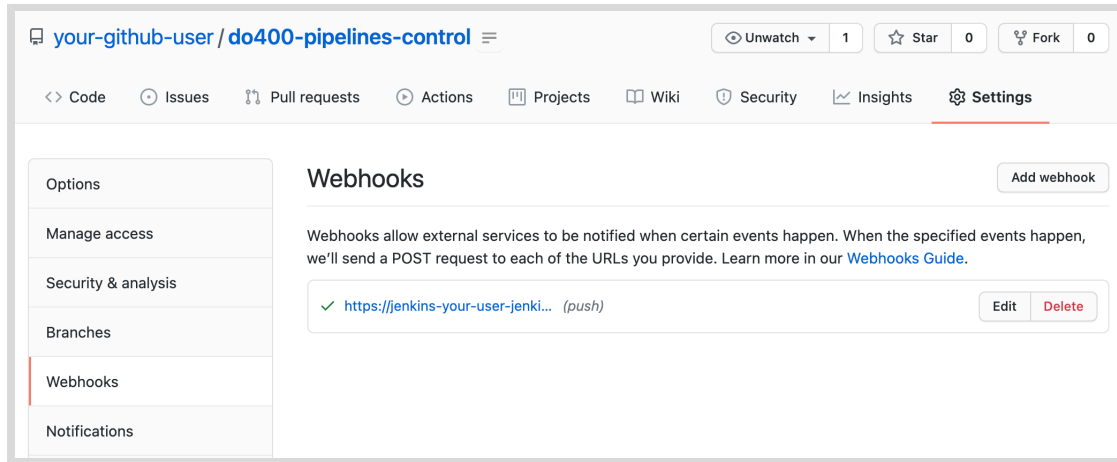
- 4.12. From the **Credentials** list select **GITHUB_TOKEN** and click **Test** connection to verify that the personal token has access to GitHub. Select the **Manage hooks** check box. Click **Apply** to save the changes made to the Jenkins configuration.
- 4.13. Click **Advanced**, and then click **Re-register hooks for all jobs**. Click **Save** to save the configuration and return to the Jenkins homepage.



- 4.14. Return to the GitHub tab and navigate to the GitHub repository of the guided exercise.

The URL of the repository follows the pattern `https://github.com/YOUR_GITHUB_USER/do400-pipelines-control`.

- 4.15. Click the **Settings** tab, and then click **Webhooks** to verify that the repository has a webhook with a green tick. The green tick indicates that the connection between Jenkins and GitHub is successful.



With this integration, any change in the repository will be notified to the Jenkins server.

- 4.16. Close the GitHub tab in your browser.

► **5.** Update the pipeline and transform it into a declarative one.

- 5.1. Open the `Jenkinsfile` file and transform the scripted pipeline into a declarative one. The code should look like the following:

```
pipeline {
  agent {
    node {
      label 'nodejs'
    }
  }
  stages {
    stage('Backend Tests') {
      steps {
        sh 'node ./backend/test.js'
      }
    }
    stage('Frontend Tests') {
      steps {
        sh 'node ./frontend/test.js'
      }
    }
  }
}
```

Notice the difference in syntax between the scripted and the declarative pipelines. Also notice that in declarative pipelines there is no need for a checkout stage because that step is already implicit.

- 5.2. Stage and commit the changes made to the `Jenkinsfile` file to the local repository.

```
[user@host pipelines-control]$ git add Jenkinsfile
[user@host pipelines-control]$ git commit -m "Using a declarative pipeline"
[main 40fa1fb] Using a declarative pipeline
1 file changed, 16 insertions(+), 9 deletions(-)
```

- 5.3. Push the new pipeline to the remote repository with the `git push` command.

```
[user@host pipelines-control]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-pipelines-control.git
f4db9c5..40fa1fb HEAD -> main
```

- 5.4. Return to the browser window and refresh the page. In the pane to the right, notice the blinking blue circle in the Jenkins jobs table. The blinking circle indicates that a build is running in that project.
- 5.5. In the jobs table, click **do400-pipelines-control** to navigate to the pipeline details page. Wait for the build to finish and verify that it is successful.
Notice that the **Build History** contains two build items. The first one was created when the pipeline was executed manually. The second one was created when we pushed changes to the repository from the command line terminal.

► 6. Parallelize stages to reduce the amount of time for the pipeline to finish.

- 6.1. Open the `Jenkinsfile` file and parallelize the testing steps. The code should look like the following:

```
pipeline {
  ...output omitted...
  stages {
    stage('Run Tests') {
      parallel {
        stage('Backend Tests') {
          steps {
            sh 'node ./backend/test.js'
          }
        }
        stage('Frontend Tests') {
          steps {
            sh 'node ./frontend/test.js'
          }
        }
      }
    }
  }
}
```

Running these groups of stages in parallel is possible due to a lack of dependencies between them.

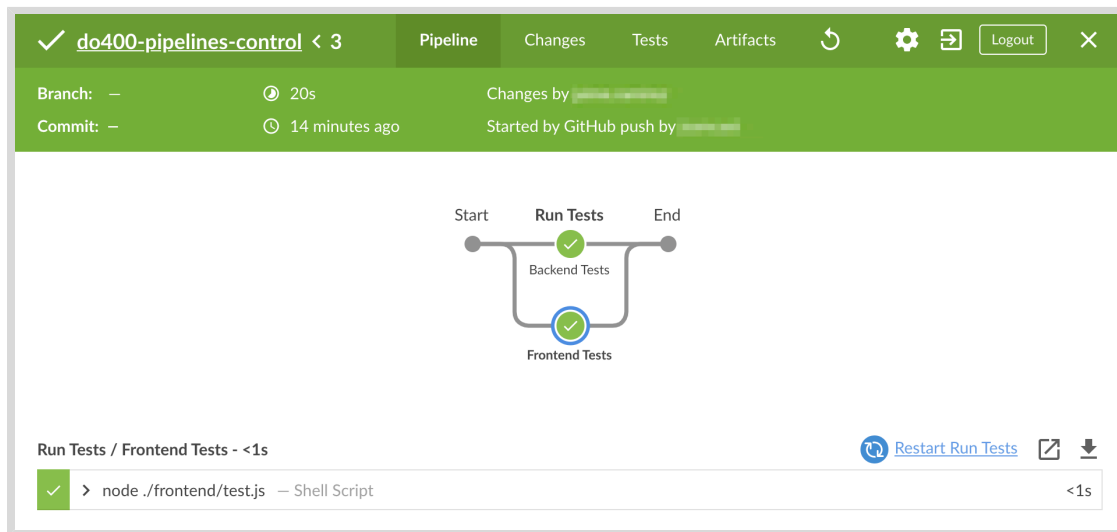
- 6.2. Stage and commit the changes made to the `Jenkinsfile` file to the local repository.

```
[user@host pipelines-control]$ git add Jenkinsfile
[user@host pipelines-control]$ git commit -m "Parallelizing steps"
[main c632771] Parallelizing steps
1 file changed, 23 insertions(+), 19 deletions(-)
rewrite Jenkinsfile (63%)
```

- 6.3. Push the updated pipeline to the remote repository with the `git push` command.

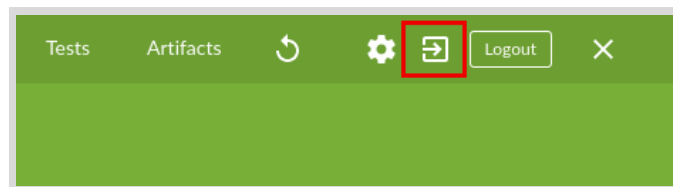
```
[user@host pipelines-control]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-pipelines-control.git
40fa1fb..c632771 HEAD -> main
```

- 6.4. Return to the browser window. Click **Open Blue Ocean** in the left pane for an alternative visualization of the current pipeline.
- The Blue Ocean interface simplifies the visualization of the pipeline and uses a two-dimensional figure to represent the different stages.
- 6.5. In the table containing previously-run builds, click the first one to see the build details. Wait for the build to finish. Notice that the tests steps are represented as parallel steps instead of serial ones.



The Blue Ocean interface displays all the stages that are running and allow you to see the output for each one of the stages.

- 6.6. Click the exit icon in the top right to return to the classic Jenkins interface.



- 6.7. Click **Back to Project** in the left pane to return to the pipeline details page.

You can return to the Blue Ocean interface at any moment by clicking **Open Blue Ocean** in the left pane.

- ▶ 7. Add a conditional step to run or skip a stage depending on the value of a boolean parameter.

7.1. Open the `Jenkinsfile` file and add a boolean parameter. The code should look like the following:

```
pipeline {
  agent {
    ...output omitted...
  }
  parameters {
    booleanParam(name: "RUN_FRONTEND_TESTS", defaultValue: true)
  }
  stages {
    stage('Run Tests') {
    ...output omitted...
```

The preceding code defines a boolean parameter named `RUN_FRONTEND_TESTS` with `true` as the default value. Providing a default value makes our pipeline more robust by not requiring a parameter in all cases.

- 7.2. Add a `when` directive in the `Frontend Tests` stage.
- The code should look like the following:

```
...output omitted...
stage('Frontend Tests') {
  when { expression { params.RUN_FRONTEND_TESTS } }
  steps {
    sh 'node ./frontend/test.js'
  }
}
...output omitted...
```

The `when` directive evaluates the value of the `RUN_FRONTEND_TESTS` parameter. If `RUN_FRONTEND_TESTS` is `true`, then the stage is executed.

- 7.3. Stage and commit the changes made to the `Jenkinsfile` file to the local repository.

```
[user@host pipelines-control]$ git add Jenkinsfile
[user@host pipelines-control]$ git commit -m "Added conditional step"
[main 147d70a] Added conditional step
1 file changed, 4 insertions(+)
```

- 7.4. Push the updated pipeline to the remote repository with the `git push` command.

```
[user@host pipelines-control]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-pipelines-control.git
40fa1fb..147d70a HEAD -> main
```

- 7.5. Return to the browser window. Notice that the push to GitHub scheduled a pipeline run. Wait for the build to finish and check that all the pipeline stages are executed successfully.

The **Frontend Tests** stage was executed because of the default value of the `RUN_FRONTEND_TESTS` parameter.

- 7.6. Visualize the pipeline execution in Jenkins and observe that both are executed because of the default values.

- 7.7. Click **Build with Parameters** in the left pane.

The current pipeline of the application requires parameters and, on manual builds, you can select the values you want to use for a specific build run.



Note

When a pipeline includes parameters, the **Build Now** button in the left pane is transformed into a **Build with Parameters** button. The transformation only occurs when the pipeline details page is refreshed.

- 7.8. Clear the `RUN_FRONTEND_TESTS` check box and click **Build** to schedule a new pipeline run. Wait for the build to finish and check that the **Frontend Tests** stage was not executed.

- ▶ 8. Add a deployment stage to be only executed when the pipeline is executed in the **main** branch.

- 8.1. Open the `Jenkinsfile` file and add stage named **Deploy** after the **Run tests** stage. The **Deploy** stage simulates the deployment of the application to an environment and it only echoes a string. The code should look like the following:

```
pipeline {
    ...output omitted...
    stages {
        stage('Run Tests') {
            ...output omitted...
        }
        stage('Deploy') {
            when {
                expression { env.GIT_BRANCH == 'origin/main' }
            }
            steps {
                echo 'Deploying...'
            }
        }
    }
}
```



Note

In multibranch pipelines you can replace the preceding `when` expression with: `when { branch 'main' }`.

- 8.2. Stage and commit the changes made to the `Jenkinsfile` file to the local repository.

```
[user@host pipelines-control]$ git add Jenkinsfile
[user@host pipelines-control]$ git commit -m "Added deploy stage"
[main c2aea59] Added deploy stage
1 file changed, 8 insertions(+)
```

- 8.3. Push the updated pipeline to the remote repository with the `git push` command.

```
[user@host pipelines-control]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-pipelines-control.git
40fa1fb..c2aea59 HEAD -> main
```

- 8.4. Return to the browser window and wait for the build to finish. Notice that all stages are executed, including the recently added `Deploy` stage.

- 8.5. Return to the command line terminal and create a `stage` branch.

```
[user@host pipelines-control]$ git checkout -b stage
Switched to a new branch 'stage'
```

- 8.6. Open the `Jenkinsfile` file and change the message printed in the `Deploy` stage. The code should look like the following:

```
...output omitted...
stage('Deploy') {
    when {
        expression { env.GIT_BRANCH == 'origin/main' }
    }
    steps {
        echo 'Step not executed...'
    }
}
...output omitted...
```

- 8.7. Stage and commit the changes made to the `Jenkinsfile` file to the local repository.

```
[user@host pipelines-control]$ git add Jenkinsfile
[user@host pipelines-control]$ git commit -m "Updated deploy message"
[main e614340] Updated deploy message
1 file changed, 1 insertion(+), 1 deletion(-)
```

- 8.8. Push the updated pipeline to the remote with the `git push` command.

```
[user@host pipelines-control]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-pipelines-control.git
* [new branch] HEAD -> stage
```

8.9. Return to the browser window and wait for the build to finish. Notice that the **Deploy** stage is not executed.

► 9. Allow the pipeline to be paused and let Jenkins users interact with the build.

9.1. Return to the command line terminal and move to the `main` branch.

```
[user@host pipelines-control]$ git checkout main
Switched to branch 'main'
```

9.2. Open the `Jenkinsfile` file and add an input step to the deployment stage. The code should look like the following:

```
...output omitted...
stage('Deploy') {
    when {
        expression { env.GIT_BRANCH == 'origin/main' }
        beforeInput true
    }
    input {
        message 'Deploy the application?'
    }
    steps {
        echo 'Deploying...'
    }
}
...output omitted...
```

The `input` directive pauses the pipeline execution and allows Jenkins users to interact with the build. When the `input` and the `when` directives are defined in a stage, Jenkins executes first the `input` directive. Adding the `beforeInput` option to the `when` directive changes the order in which Jenkins evaluates the directives.

In the preceding code the `input` directive is only executed when the pipeline runs in the `main` branch.

9.3. Stage and commit the changes made to the `Jenkinsfile` file to the local repository.

```
[user@host pipelines-control]$ git add Jenkinsfile
[user@host pipelines-control]$ git commit -m "Added input"
[main 23e085f] Added input
1 file changed, 4 insertions(+)
```

9.4. Push the updated pipeline to the remote with the `git push` command.

```
[user@host pipelines-control]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-pipelines-control.git
c5ae622..23e085f HEAD -> main
```

9.5. Return to the browser window and notice that the pipeline execution is paused in the **Deploy** stage.

- 9.6. Click the current build number or date in the **Build History** area to go to the build details page. Click **Paused for input** in the left pane, and then click **Abort** to stop the pipeline execution.
When submitting the form, Jenkins redirects you to the **Console Output** page.
- 9.7. Wait for the build to finish and check in the console output that the final result of the job is **ABORTED**.
- 9.8. Click **Back to project** in the left pane to navigate to the pipeline details page. Notice in the **Stage View** area that the last pipeline run executed successfully all the stages except the one named **Deploy**.
- 9.9. Click **Build with Parameters** in the left pane to schedule a pipeline run. Keep the **RUN_FRONTEND_TESTS** check box selected and click **Build**.
- 9.10. Wait until the pipeline pauses in the **Deploy** stage.
- 9.11. Hover with your mouse the running **Deploy** stage, and then on the pop-up menu, click **Proceed** to continue the pipeline execution.
- 9.12. Wait for the build to finish and check that the build was successful.
- 9.13. Click **Back to project** in the left pane to navigate to the pipeline details page. Notice in the **Stage View** area that the last pipeline run marked the **Deploy** stage as successful.
- ▶ **10.** Clean up. Click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.

This concludes the guided exercise.

► Lab

Authoring Pipelines

In this lab, you will build a basic Jenkins pipeline for a sample Quarkus application.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (`) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Create a Jenkins pipeline by using a declarative approach
- Use conditionals to avoid executing stages
- Parallelize stages
- Integrate your pipeline with a GitHub repository

Before You Begin

To perform this lab, ensure you have:

- Git installed
- A GitHub account
- Access to a Jenkins instance
- The `D0400-apps` repository cloned in your workspace folder
- A web browser such as Firefox or Chrome
- Access to a command-line terminal



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

The `shopping-cart-v2` application that you will use in this exercise is a simple Quarkus application simulating the behavior of a shopping cart.

The application exposes an API with several endpoints. It is a CRUD application, which allows you to read, add, update and remove products from a shopping cart.

1. Create a new public GitHub repository containing the files from `D0400-apps/shopping-cart-v2`. Use `main` as the default branch of the repository.
2. In your Jenkins instance, create a new pipeline named `D0400-pipelines-lab` and link it to the GitHub repository you have previously created. Use a `Multibranch pipeline`.
3. In your local environment, create a new file named `Jenkinsfile`. The pipeline must run in any available agent. Use a declarative syntax.

Create a stage named `Test` where two nested stages are executed in parallel, `Unit tests` and `Integration tests`.

For the `Unit tests` stage, use the command `./mvnw test -D testGroups=unit`.

For the `Integration tests` stage, use the command `./mvnw test -D testGroups=integration`. Include also a `RUN_INTEGRATION_TESTS` boolean parameter, which will determine whether the integration tests stage should be executed (the default value for this parameter should be `true`).

4. Commit and push the changes to your GitHub repository. Verify in your Jenkins instance that the pipeline runs successfully and executes the `Test` stage.
5. After the `Test` stage, add a stage named `Build`, which creates a JAR file (Java executable) from the source code. Use the Maven command `./mvnw package -D skipTests` to obtain the JAR file.
If an error occurs when creating the executable, print the message `Error while generating JAR file` in the Jenkins console. Use a `script` step with a `try/catch` block from the Groovy programming language.
6. Commit and push the changes to your GitHub repository. Verify in your Jenkins instance that the pipeline runs successfully and executes the `Test` and `Build` stages.
7. Clean up. Remove the Jenkins pipeline and the GitHub project you have created for this lab.

This concludes the lab.

► Solution

Authoring Pipelines

In this lab, you will build a basic Jenkins pipeline for a sample Quarkus application.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (`) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Create a Jenkins pipeline by using a declarative approach
- Use conditionals to avoid executing stages
- Parallelize stages
- Integrate your pipeline with a GitHub repository

Before You Begin

To perform this lab, ensure you have:

- Git installed
- A GitHub account
- Access to a Jenkins instance
- The `DO400-apps` repository cloned in your workspace folder
- A web browser such as Firefox or Chrome
- Access to a command-line terminal



Important

Try to keep your course work organized by creating a workspace folder, such as `~/DO400`. Start the guided exercise from this folder.

Instructions

The `shopping-cart-v2` application that you will use in this exercise is a simple Quarkus application simulating the behavior of a shopping cart.

The application exposes an API with several endpoints. It is a CRUD application, which allows you to read, add, update and remove products from a shopping cart.

1. Create a new public GitHub repository containing the files from `D0400-apps/shopping-cart-v2`. Use `main` as the default branch of the repository.
 - 1.1. Open a web browser and navigate to `http://github.com`. Sign in with your GitHub credentials.
 - 1.2. Create a new public, empty repository and name it `D0400-pipelines-lab`.
 - 1.3. Open a command-line terminal, navigate to the `D0400-apps` application folder, and checkout the `main` branch to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
...output omitted...
```

- 1.4. Copy the contents of the `D0400-apps/shopping-cart-v2` folder to the `pipelines-lab` folder. Navigate to the new directory.

```
[user@host D0400-apps]$ cp -Rv shopping-cart-v2/ \
~/D0400/pipelines-lab
...output omitted...
[user@host D0400-apps]$ cd ../pipelines-lab
```



Important

Windows users must replace the preceding `cp` command in PowerShell as follows:

```
PS C:\Users\user\D0400\D0400-apps> Copy-Item -Path shopping-cart-v2 `
>> -Destination ~\D0400\pipelines-lab -Recurse
```

- 1.5. Initialize a Git repository, commit all files and set up a remote.

```
[user@host pipelines-lab]$ git init
...output omitted...
[user@host pipelines-lab]$ git add .
```



Important

Windows users must also execute the following command after the `git add` command:

```
PS C:\Users\user\D0400\pipelines-lab> git update-index --chmod=+x mvnw
```

```
[user@host pipelines-lab]$ git commit -m "Initial commit"
...output omitted...
[user@host pipelines-lab]$ git branch -M main
[user@host pipelines-lab]$ git remote add origin \
https://github.com/YOUR_GITHUB_USER/D0400-pipelines-lab.git
[user@host pipelines-lab]$ git push --set-upstream origin main
...output omitted...
```

Replace YOUR_GITHUB_USER with your actual GitHub user.

2. In your Jenkins instance, create a new pipeline named `D0400-pipelines-lab` and link it to the GitHub repository you have previously created. Use a **Multibranch pipeline**.
 - 2.1. Open a new tab in the browser and navigate to the URL of the Jenkins server, which you installed in the first guided exercise. If prompted, log into Jenkins with your Red Hat OpenShift credentials.
 - 2.2. Click **New item** in the left navigation pane.
 - 2.3. Enter `D0400-pipelines-lab` as the item name. Click **Multibranch Pipeline** and then click **OK**.
 - 2.4. In the **Branch Sources** section, click **Add source** and select **Git**.
 - 2.5. In the **Project Repository** field include the URL of the GitHub repository created previously.
The URL of the repository looks like `https://github.com/YOUR_GITHUB_USER/D0400-pipelines-lab.git`
 - 2.6. Click **Save**. Verify that the scanning of the pipeline finishes successfully and no `Jenkinsfile` file is found. The **Scan Multibranch Pipeline** Log output should look like:

```
...output omitted...
Checking branches...
  Checking branch main
    'Jenkinsfile' not found
...output omitted...
Finished: SUCCESS
```

3. In your local environment, create a new file named `Jenkinsfile`. The pipeline must run in any available agent. Use a declarative syntax.
Create a stage named **Test** where two nested stages are executed in parallel, **Unit tests** and **Integration tests**.
For the **Unit tests** stage, use the command `./mvnw test -D testGroups=unit`.
For the **Integration tests** stage, use the command `./mvnw test -D testGroups=integration`. Include also a `RUN_INTEGRATION_TESTS` boolean parameter, which will determine whether the integration tests stage should be executed (the default value for this parameter should be `true`).
 - 3.1. Create a new file named `Jenkinsfile`. Add a **pipeline** block. Inside **pipeline**, include an **agent** section with the parameter **any**. Add a **stage** directive with the name **Test**.

```

pipeline {
    agent any

    stages {
        stage('Test') {

        }
    }
}

```

3.2. Add a `parallel` section to execute nested stages.

```

pipeline {
    agent any

    stages {
        stage('Test') {
            parallel {

            }
        }
    }
}

```

3.3. Append a new stage named `Unit tests`. Use the `sh` step to run the Maven command `./mvnw test -D testGroups=unit`.

```

pipeline {
    agent any

    stages {
        stage('Test') {
            parallel {
                stage('Unit tests') {
                    steps {
                        sh './mvnw test -D testGroups=unit'
                    }
                }
            }
        }
    }
}

```

3.4. Add another stage named `Integration tests` that executes the command `./mvnw test -D testGroups=integration`. Declare a boolean parameter named `RUN_INTEGRATION_TESTS` by using the `parameters` directive. Set `true` as the default value for the parameter.

Include a `when` directive to execute this stage when `RUN_INTEGRATION_TESTS` parameter is `true`.

```

pipeline {
    agent any

    parameters {
        booleanParam(name: "RUN_INTEGRATION_TESTS", defaultValue: true)
    }

    stages {
        stage('Test') {
            parallel {
                ...output omitted...

                stage('Integration tests') {
                    when {
                        expression { return params.RUN_INTEGRATION_TESTS }
                    }

                    steps {
                        sh './mvnw test -D testGroups=integration'
                    }
                }
            }
        }
    }
}

```

4. Commit and push the changes to your GitHub repository. Verify in your Jenkins instance that the pipeline runs successfully and executes the Test stage.

- 4.1. Add, commit, and push the Jenkinsfile file.

```

[user@host pipelines-lab]$ git add Jenkinsfile
[user@host pipelines-lab]$ git commit -m "Add Test stage to Jenkinsfile"
...output omitted...
[user@host pipelines-lab]$ git push
...output omitted...

```

- 4.2. Return to the Jenkins tab in your browser. In the left pane, click **Scan Multibranch Pipeline Now** and then click **Status**. Click the **main** branch in the **Branches** area.
- 4.3. Wait for a new build to appear in the **Build history** section. Navigate to the build details page.
- 4.4. Click **Console output** in the left pane, then verify that unit and integration tests run in parallel, and finally verify that the build finishes successfully.



Note

When the console output is too large, Jenkins skips some of it. Click **Full Log** in the top of the **Console Output** area to show the entire log content.

The output should look similar to this:


```

...output omitted...
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] parallel
[Pipeline] { (Branch: Unit tests)
[Pipeline] { (Branch: Integration tests)
[Pipeline] stage
[Pipeline] { (Unit tests)
[Pipeline] stage
[Pipeline] { (Integration tests)
[Pipeline] sh
[Pipeline] sh
+ ./mvnw test -D testGroups=unit
Picked up JAVA_TOOL_OPTIONS: -XX:+UnlockExperimentalVMOptions -
Dsun.zip.disableMemoryMapping=true
+ ./mvnw test -D testGroups=integration
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS

```

The output of unit and integration tests is mixed up in the console because they run at the same time.

You have not provided any value for the `RUN_INTEGRATION_TESTS` parameter therefore Jenkins has used the default value `true`.

- 4.5. Click **Back to Project** in the left pane and then click **Build with Parameters**. Clear the `RUN_INTEGRATION_TESTS` check box and click **Build**. Wait for a new build to appear in the **Build history** area, and navigate to the build details page.
- 4.6. Click **Console output** in the left pane, then verify that Jenkins does not execute the **Integration tests** stage, and finally verify that the build finishes successfully.

The output should look similar to this:

```

...output omitted...
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] parallel
[Pipeline] { (Branch: Unit tests)
[Pipeline] { (Branch: Integration tests)
[Pipeline] stage
[Pipeline] { (Unit tests)
[Pipeline] stage
[Pipeline] { (Integration tests)
Stage "Integration tests" skipped due to when conditional
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] sh
+ ./mvnw test -D testGroups=unit
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS

```

5. After the Test stage, add a stage named Build, which creates a JAR file (Java executable) from the source code. Use the Maven command `./mvnw package -D skipTests` to obtain the JAR file.

If an error occurs when creating the executable, print the message **Error while generating JAR file** in the Jenkins console. Use a script step with a try/catch block from the Groovy programming language.

- 5.1. Add a new stage Build with a steps section inside. Within the steps section, include a script step.

```
pipeline {
    ...output omitted...

    stages {
        ...output omitted...

        stage('Build') {
            steps {
                script {

                }
            }
        }
    }
}
```

- 5.2. Add a try/catch block inside script.

In the try part, include a sh step that executes the `./mvnw package -D skipTests` command.

In the catch part, include an echo step to send the **Error while generating JAR file** message to the standard output. Throw the exception again to mark the pipeline as failed.

```
pipeline {
    ...output omitted...

    stages {
        ...output omitted...

        stage('Build') {
            steps {
                script {
                    try {
                        sh './mvnw package -D skipTests'
                    } catch (ex) {
                        echo "Error while generating JAR file"
                        throw ex
                    }
                }
            }
        }
    }
}
```

6. Commit and push the changes to your GitHub repository. Verify in your Jenkins instance that the pipeline runs successfully and executes the **Test** and **Build** stages.

6.1. Add, commit, and push the `Jenkinsfile` file.

```
[user@host pipelines-lab]$ git add Jenkinsfile
[user@host pipelines-lab]$ git commit -m "Add Build stage to Jenkinsfile"
...output omitted...
[user@host pipelines-lab]$ git push
...output omitted...
```

6.2. Return to the Jenkins tab in your browser. Click **Back to Project** in the left pane.

6.3. Click **Build with Parameters** and then click **Build**. Wait for a new build to appear in the **Build history** area, and navigate to the build details page.

6.4. Click **Console output** in the left pane. Verify that the build finishes successfully and Jenkins executes both **Test** and **Build** stages.

The output should look similar to this:

```
...output omitted...
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] parallel
[Pipeline] { (Branch: Unit tests)
[Pipeline] { (Branch: Integration tests)
[Pipeline] stage
[Pipeline] { (Unit tests)
[Pipeline] stage
[Pipeline] { (Integration tests)
[Pipeline] sh
[Pipeline] sh
+ ./mvnw test -D testGroups=unit
Picked up JAVA_TOOL_OPTIONS: -XX:+UnlockExperimentalVMOptions -
Dsun.zip.disableMemoryMapping=true
+ ./mvnw test -D testGroups=integration
...output omitted...
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ ./mvnw package -D skipTests
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS
```

7. Clean up. Remove the Jenkins pipeline and the GitHub project you have created for this lab.

7.1. From the **Console output** page, click **Back to Project** and then click **Up**.

7.2. Click **Delete Multibranch Pipeline** and then click **Yes**.

You have removed the Jenkins pipeline.

- 7.3. In your browser, navigate to the GitHub repository you have used in this lab.
The URL should look like `https://github.com/YOUR_GITHUB_USER/DO400-pipelines-lab`.
- 7.4. Click **Settings**. In the **Danger Zone** area at the bottom of the page, click **Delete this repository**.
- 7.5. In the confirmation message window, you must include the name of the repository `YOUR_GITHUB_USER/DO400-pipelines-lab`. Click **I understand the consequences, delete this repository**.
You have removed the GitHub repository.

This concludes the lab.

Summary

In this chapter, you learned:

- A pipeline is a series of connected steps executed in sequence to accomplish a task.
- A step is a single task to perform inside a project, and stages are groups of steps.
- A project in Jenkins describes an executable workflow, and a build is the result of a pipeline execution.
- Declarative pipelines have a strict and predefined structure.
- Scripted pipelines have a flexible structure, and they are written in Groovy.
- You can define scripted blocks inside declarative pipelines.
- Jenkins has three types of stage executions: sequential, parallel, and matrix.
- You can control the step execution by using parameters, checkpoints, and conditionals.

Chapter 6

Deploying Applications with Pipelines

Goal

Safely and automatically deploy applications to OpenShift Container Platform.

Objectives

- Create application images and deploy applications with pipelines.
- Use checkpoints to deploy to various environments.
- Release applications by using various strategies to mitigate downtime and risk.
- Configure pipelines to automatically test an application to lower risk for production outages.

Sections

- Building Images and Deploying to OpenShift (and Guided Exercise)
- Deploying Applications to Various Environments (and Guided Exercise)
- Implementing Release Strategies (and Guided Exercise)
- Executing Automated Tests in Pipelines (and Guided Exercise)

Lab

Deploying Applications with Pipelines

Building Images and Deploying to OpenShift

Objectives

After completing this section, you should be able to create application images and deploy applications with pipelines.

Building and Deploying with OpenShift

In previous chapters, you learned how to organize and test code, as well as how to automate those processes by using Jenkins. In this chapter, you will learn the basics of *continuous delivery* by using Jenkins and OpenShift.

As part of continuous delivery, you must be able to automatically deploy your applications at the push of a button. See *Defining CI/CD and DevOps* for a more robust explanation of continuous delivery, especially as it compares to *continuous deployment*.

Container orchestration platforms, such as Kubernetes and OpenShift, aid these deployment processes by making them more consistent, scalable, and follow a pull-based approach by using image promotion.

In the corresponding guided exercise, you will use Jenkins with OpenShift to automate building and deploying an example application.

Packaging Applications

To deploy your application, you must first compile and package it. How you package your code depends on your chosen language and libraries, but most follow a similar structure.

Packaging systems produce one or a few files, generally called *artifacts*, which are used by your deployment system. For example, Maven generates artifact files based on the Java Archive (JAR) format.

Note that artifacts can be nested, and one artifact might contain one or more other artifacts. For example, a Java Web Archive (WAR) often contains several JAR files, and an Enterprise Application Archive (EAR) might contain JAR and WAR files.

Artifact Repositories

Artifact repositories are systems and services that enable uploading, storing, and retrieving artifacts. A given artifact repository might manage a single artifact type or a combination of several.

Examples of common artifact repositories:

- Sonatype Nexus
- Artifactory by JFrog
- Quay.io and Docker Hub for container images

Jenkins can also directly store artifacts built as part of a pipeline. This might be all you need, if you have minimal artifact storage needs. More detail can be found in *Deploying Applications to Various Environments*.

Managing Container Images as Artifacts

Containers encapsulate the runtime dependencies and environment for an application and enable multiple applications to share the same host operating system. You run containers by using specialized software, called a *container engine*, such as Podman, Docker, and CRI-O.

A container is instantiated from a corresponding *container image*, which is made up of layers containing pieces of a file system. These container images are a type of artifact. Consisting of file system layers, they can include any amount of file-based data. They often contain other types of artifacts, such as JAR files.

However, they differ from file-based artifacts in how they are transmitted and stored. Instead of a file representation and transmission, they rely on the container engine itself for uploading and downloading. Because of this, artifact repositories need special support for container images.

Container Image Registries

An *image registry* is a type of artifact repository that can manage container images. For example, one such registry is Quay.io, which you will use throughout the rest of this course.

Push images to Quay via the following:

- Manually, via a command-line interface, with a tool such as Podman.
- Automatically, via CI tooling such as a Jenkins pipeline or with a triggered build in OpenShift.

Image Streams

When a container image is built, it can be assigned an *image tag*. These tags are human-readable strings used to reference specific versions of a container image.

In OpenShift, one or more of these tags can then be used to compose an *image stream*. Image streams are a mechanism to publish new versions of a container image as they are available. These streams are observable by a container orchestration platform in order to trigger a new deployment when a new image is available.

Deploying Container Images

A typical pipeline includes the following deployment steps:

1. Build the code, if needed.
2. Package code as one or more artifacts.
3. Create a container image with the artifacts.
4. Deploy the container image to a testing environment.
5. Perform environment-specific verification tasks.
6. Promote the image to the next environment.
7. Repeat 5-6 until the image is deployed in production.

Image Promotion

Image promotion is when a *single* image is built and deployed to *lower environments*, such as DEV and TEST. When that image is verified in those environments, it is deployed to *higher environments*, such as PRE-PROD and PROD. This is the recommended deployment pattern because creating artifacts for each environment increases the number of potential build and deploy errors. Additionally, these errors can be much harder to troubleshoot.

Image promotion also makes recreating bugs easier, because lower environments are built from the same artifacts as production. Oftentimes, the relevant artifact might already be deployed in the lower environments.



Note

The names and number of environments in your organization might differ from those used in this course. For example, many organizations maintain an additional *User-Acceptance Testing* (UAT) environment.

For simplicity, this course adheres to the following four environments, which are in ascending order of stability:

- **LOCAL:** An experimentation sandbox for developers. This can be either a local environment or a **SANDBOX** environment on a specific platform like OpenShift.
- **DEV:** An environment where developers experiment their codes merged and integrated with the others'
- **TEST:** A semi-stable environment for performing basic automated tests
- **PRE-PROD:** A replica of PROD primarily for troubleshooting bugs in PROD
- **PROD:** The production environment, which is available to users and should be as stable as possible

Role-Based Access Control

You must grant additional permissions to Jenkins for it to perform certain actions in your OpenShift project.

For example, in the corresponding guided exercise, you will grant the `edit` permission to the Jenkins service account via the following:

```
[user@host greeting-console]$ oc policy add-role-to-user \
edit system:serviceaccount:your-jenkins-project:jenkins \
-n your-project
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"
```



Note

You will need the `oc` command installed and configured on your system to view and manage OpenShift roles and permissions. Installation instructions can be found in *Guided Exercise: Configuring a Developer Environment*.

Triggering Image Deployments

In a traditional environment, you often manually trigger deployments. A better solution would be to automate deployment with your CI/CD tool, such as Jenkins. However, if not done carefully, you can accidentally create race conditions due to parallel deployments. More importantly, if used for all of your deployments, your CI/CD tool can become a deployment bottleneck, slowing down your overall deployment frequency.

It is better to configure OpenShift to deploy whenever a new image is pushed to a specific image stream. This method utilizes OpenShift's advanced capabilities to orchestrate when and how deployments are performed. This also frees up your CI/CD tool to focus on other tasks, such as running tests and building artifacts.



References

Quay.io

<https://quay.io/repository/>

Artifactory

<https://jfrog.com/artifactory/>

Sonatype Nexus

<https://www.sonatype.com/nexus/>

Podman

<https://podman.io>

CRI-O

<https://cri-o.io>

Continuous Delivery vs Continuous Deployment

<https://azure.microsoft.com/en-us/overview/continuous-delivery-vs-continuous-deployment/>

Using RBAC to define and apply permissions

https://docs.openshift.com/container-platform/4.6/authentication/using-rbac.html#adding-roles_using-rbac

► Guided Exercise

Building Images and Deploying to OpenShift

In this exercise you will use Jenkins pipelines to build images, push them to an external repository, and deploy images as services in OpenShift.

You will deploy two applications:

- **greeting-console**: A simple greeting command-line application. You publish this application as a container image to a public registry.
- **greeting-service**: A simple greeting application that exposes an API. You will deploy this application as a service to OpenShift.

The solution files are provided at the `greeting-console`, and `greeting-service-v1` folders in the `solutions` branch of the `D0400-apps` repository. Notice that you might need to replace some strings in the `Jenkinsfile` files.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (```) instead of the backslash character (`\`) to introduce line breaks in these commands.

Outcomes

You should be able to use pipelines to build container images and deploy applications to OpenShift.

Before You Begin

To perform this exercise, ensure you have:

- Your `D0400-apps` fork cloned in your workspace
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- The OpenShift CLI
- An account on <https://quay.io/>



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- ▶ 1. Create a new repository to work on the `greeting-console` application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps` application folder and checkout the `main` branch to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
...output omitted...
```

- 1.2. Copy the contents of the `~/D0400/D0400-apps/greeting-console` folder to a new `greeting-console` folder in your workspace.

```
[user@host D0400-apps]$ cp -Rv greeting-console ~/D0400/greeting-console
...output omitted...
[user@host D0400-apps]$ cd ../greeting-console
[user@host greeting-console]$
```



Important

Windows users must replace the preceding `cp` command in PowerShell as follows:

```
PS C:\Users\user\D0400\D0400-apps> Copy-Item -Path greeting-console `
>> -Destination ~\D0400\greeting-console -Recurse
```

- 1.3. Create an empty public repository in GitHub named `do400-greeting-console` to host the source code of the greeting console application.
- 1.4. Initialize a local repository in the `~/D0400/greeting-console` folder and add the remote origin.

```
[user@host greeting-console]$ git init .
Initialized empty Git repository in ...output omitted.../D0400/greeting-
console/.git/
[user@host greeting-console]$ git remote \
add origin https://github.com/YOUR_GITHUB_USER/do400-greeting-console.git
```

If prompted, enter your GitHub user name, and personal access token.

- 1.5. Commit the current version of the code, set `main` as the default branch, and push it to the origin repository.

```
[user@host greeting-console]$ git add -A
[user@host greeting-console]$ git commit -m "Initial commit"
[master (root-commit) e453e03] Initial commit
...output omitted...
[user@host greeting-console]$ git branch -M main
[user@host greeting-console]$ git push origin main
...output omitted...
To https://github.com/your_github_user/do400-greeting-console.git
* [new branch]      main -> main
```

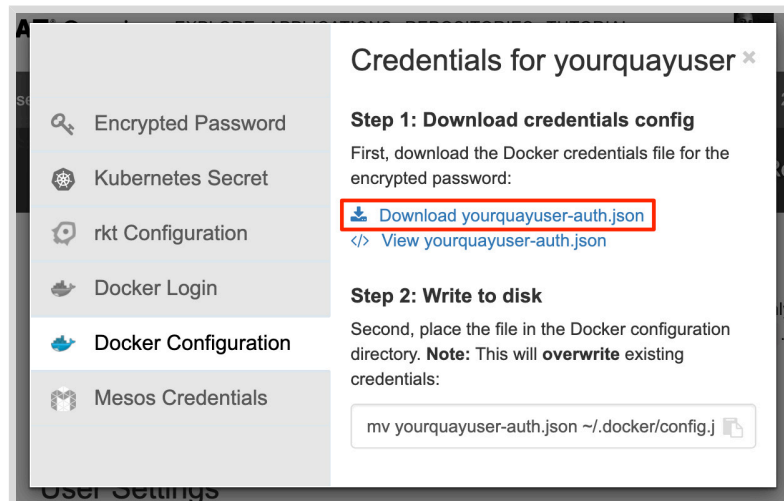
- ▶ 2. Prepare OpenShift to build and push images to your Quay account.
 - 2.1. Log in to OpenShift and ensure that you are using the `RHT_OCP4_DEV_USER-jenkins` project. To log in, use the credentials provided in the Lab Environment tab of the course's online learning website, after you provision your online lab environment. You can also copy the login command from the OpenShift web console.

```
[user@host greeting-console]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
[user@host greeting-console]$ oc project RHT_OCP4_DEV_USER-jenkins
Now using project "youruser-jenkins"
on server "https://api.cluster.example.com:6443".
```

- 2.2. Create a new OpenShift project to host the greeting applications and switch to the new project.

```
[user@host greeting-console]$ oc new-project RHT_OCP4_DEV_USER-greetings
Now using project "youruser-greetings" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 2.3. In a web browser, navigate to <https://quay.io/> and click **Sign in** in the upper right to login with your Quay account.
- 2.4. After logging in, click **yourquayuser** → **Account Settings** in the upper right.
- 2.5. In the **Docker CLI Password** section, click **Generate Encrypted Password** and enter your Quay password when prompted. In the credentials window that opens, select **Docker Configuration**. Click **Download yourquayuser-auth.json** to download the credentials file to your workstation.



**Note**

In some browsers, the download button does not display. If it does not display for you, then click **View yourquayuser-auth.json**, copy the contents, and manually paste them into a new file.

- 2.6. Use the `oc create secret` command to create a secret by using the credentials file, which you just downloaded. Specify the file path of the *yourquayuser-auth.json* file with the `from-file` parameter.

```
[user@host greeting-console]$ oc create secret generic \
quay-credentials \
--from-file=.dockerconfigjson=path/to/yourquayuser-auth.json \
--type=kubernetes.io/dockerconfigjson
secret/quay-credentials created
```

- 2.7. Configure a `BuildConfig` resource to build and push your application image to quay.io. This `BuildConfig` resource will build an image by using the Dockerfile from your `do400-greeting-console` repository and will push the image to your Quay account.

Open the `build.yml` file and specify your GitHub and Quay user names.

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: greeting-console
spec:
  nodeSelector: {}
  strategy:
    type: Docker ❶
  source:
    type: Git
    git:
      uri: 'https://github.com/YOUR_GITHUB_USER/do400-greeting-console' ❷
      ref: main
  output:
    to:
      kind: DockerImage
      name: quay.io/YOUR_QUAY_USER/greeting-console ❸
  pushSecret:
    name: quay-credentials ❹
```

- ❶ The Docker build strategy generates a container image by using a Dockerfile.
 - ❷ The repository where the Dockerfile is located. Replace `YOUR_GITHUB_USER` with your actual GitHub user name.
 - ❸ The location to push the image to. Replace `YOUR_QUAY_USER` with your actual Quay user name.
 - ❹ The name of the quay credentials secret that you previously created.
- Save the file.

2.8. Create the `BuildConfig` resource with `oc create`.

```
[user@host greeting-console]$ oc create -f build.yml
buildconfig.build.openshift.io/greeting-console created
```

▶ 3. Use Jenkins Blue Ocean to create a continuous integration pipeline for your repository.

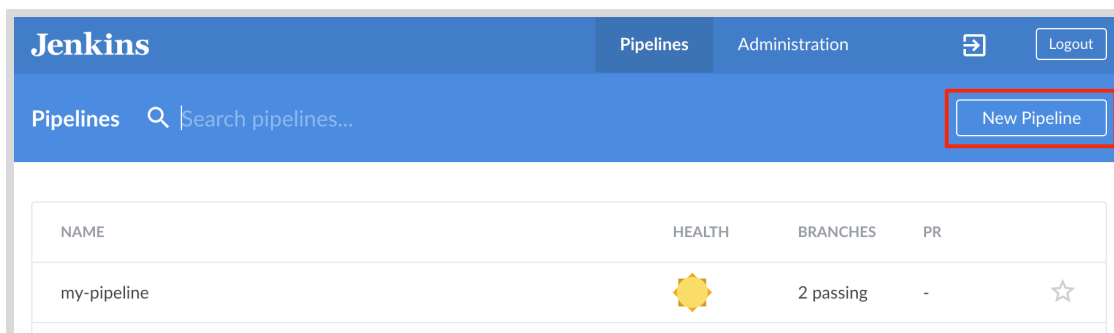
3.1. Get the URL of your Jenkins instance. Replace `RHT_OCP4_DEV_USER` with the value provided in the `Lab Environment` tab of the course's online learning website.

```
[user@host greeting-console]$ oc get route \
-n RHT_OCP4_DEV_USER-jenkins jenkins \
-o jsonpath="{.spec.host}"
jenkins-yourrhtuser-jenkins.apps.cluster.example.com
```

3.2. In your web browser, navigate to the Jenkins URL. Log in into Jenkins with your OpenShift credentials.

3.3. Click **Open Blue Ocean** in the left navigation pane to open the Blue Ocean UI.

3.4. In the Blue Ocean main page, click **Create a new Pipeline**.



3.5. In the **Where do you store your code?** step, select **GitHub**.

3.6. If Jenkins displays the **Connect to GitHub** step, click **Create an access token here**. Otherwise, ignore this step

A GitHub window opens to create a new personal access token.

- Type `D0400 Jenkins` in the **Note** field and click **Generate token** to create the token.
- In the **Personal access tokens** page, copy the newly created personal access token.

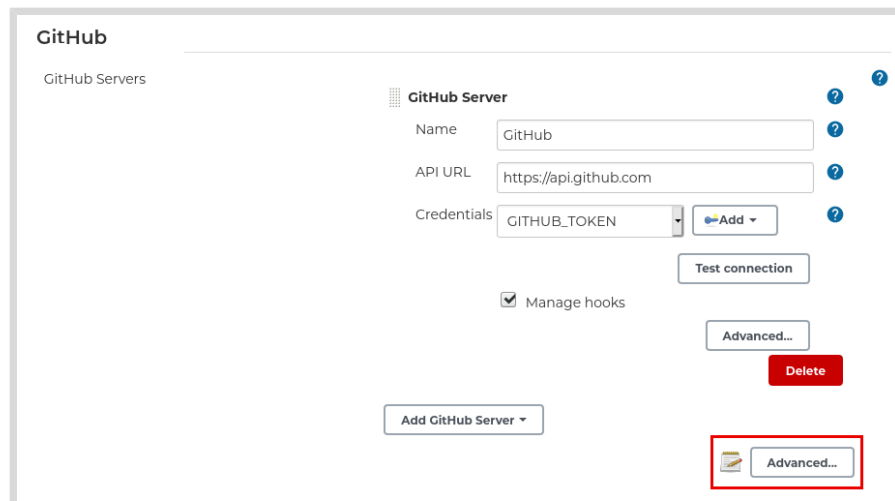


Warning

Make sure to copy the generated token. The token is only displayed once and you will need it in the following steps.

- Switch back to the Jenkins Blue Ocean browser tab.
- Paste the newly created access token in the **Your GitHub access token** input field and click **Connect**.

- 3.7. In the **Which organization does the repository belong to?** step, select your GitHub user name.
 - 3.8. In the **Choose a repository** step, select the `do400-greeting-console` repository from the list and click **Create Pipeline**. After creating the pipeline, Jenkins triggers the first pipeline run.
 - 3.9. Wait for the pipeline to finish. Verify that the pipeline already includes two passing stages: `Check Style` and `Test`.
- 4. Integrate Jenkins with GitHub to run the pipeline when new changes are pushed to the repository.
- 4.1. Click **Administration** in the upper right of the window, scroll down to the **System Configuration** section, and then click **Configure System** in the right pane.
 - 4.2. Scroll down to the **GitHub** area to add a GitHub server.
 - 4.3. If you already created a GitHub server in a previous exercise, go to step 4.8.
Click the **Add GitHub Server** list, then click **GitHub Server**. Type `GitHub` in the **Name** field. In the **Credentials** group click **Add** and then select **Jenkins** to open a window, which allows you to add credentials.
 - 4.4. In the **Jenkins Credentials Provider: Jenkins** window, select `Secret` text from the **Kind** list. In the **Secret** field, paste the personal token that you previously generated in GitHub. Type `GITHUB_TOKEN` in the **ID** field. Click **Add** to store the secret in Jenkins.
 - 4.5. From the **Credentials** list select `GITHUB_TOKEN` and click **Test connection** to verify that the personal token has access to GitHub. Select the **Manage hooks** check box.
 - 4.6. Click **Apply** to save the changes made to the Jenkins configuration.
 - 4.7. Click **Advanced**, and then click **Re-register hooks for all jobs**. Click **Save** to save the configuration and return to the Jenkins homepage.



- 4.8. Navigate to the webhooks page of your repository located at `https://github.com/YOUR_GITHUB_USER/do400-greeting-console/settings/hooks`. Verify that the repository has a webhook with a green tick. The green tick

indicates a successful connection between Jenkins and GitHub. With this integration, GitHub will notify Jenkins of any change in the repository.

► 5. Add the "Release" stage to the CI pipeline of the `greeting-console` application.

- 5.1. Open the `~/DO400/greeting-console/Jenkinsfile` and add a new stage called `Release`. The new stage starts a new build by using the `BuildConfig` resource that you created before. This build generates the image and pushes it to your Quay account.

The `Release` stage selects the `greetings` project and triggers the build. The `--follow` parameter shows the build logs. The `--wait` parameter waits for the build to finish and returns an exit code, which corresponds to the build result.

```
stage('Release') {
  steps {
    sh '''
      oc project RHT_OCP4_DEV_USER-greetings
      oc start-build greeting-console --follow --wait
    '''
  }
}
```

Replace `RHT_OCP4_DEV_USER` with your lab user to switch to your `greetings` project. Save the file.

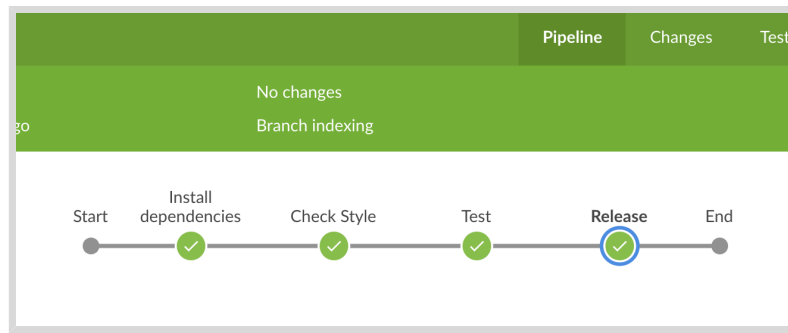
- 5.2. To execute the `Release` stage, Jenkins needs permission to start builds in the `greetings` project. Allow this by adding the `edit` role to the Jenkins account in the `RHT_OCP4_DEV_USER-greetings` project.

```
[user@host greeting-console]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"
```

- 5.3. Commit and push the changes.

```
[user@host greeting-console]$ git commit -a -m "Add release stage" *
[main 0b67344] Add release stage
 2 files changed, 10 insertions(+), 3 deletions(-)
[user@host greeting-console]$ *git push origin main
To https://github.com/your_github_user/do400-greeting-console.git
...output omitted... main -> main
```

- 5.4. Verify that the pipeline runs after pushing the changes. In your web browser, navigate to the Jenkins main page and click **Open Blue Ocean** in the left navigation pane. In the Blue Ocean main page, click `do400-greeting-console`. In the `do400-greeting-console` activity page, click the most recent build.



Click the **Release** stage and then click the **Shell script** step to show the output of the **Release** stage. Wait for the stage to finish. Inspect the logs and verify that the image has been built and pushed to Quay.

```
...output omitted...
STEP 1: FROM registry.access.redhat.com/ubi8/nodejs-12
STEP 2: COPY package.json package-lock.json /app/
...output omitted...
Successfully pushed quay.io/yourquayuser/greeting-console@...output omitted...
...output omitted...
```

- 5.5. Navigate to the image details page located at https://quay.io/repository/YOUR_QUAY_USER/greeting-console. Click **Settings** in the left navigation pane. Scroll down the **Repository Visibility** section, click **Make Public** and confirm.
- ▶ 6. Create a new repository to work on the **greeting-service** application. You will apply a continuous deployment approach to deploy this application to OpenShift.
- 6.1. Change to your workspace folder, then copy the contents of the `~/D0400/D0400-apps/greeting-service` folder to a new **greeting-service** folder in your workspace.

```
[user@host greeting-console]$ cd ~/D0400
[user@host D0400]$ cp -Rv ~/D0400/D0400-apps/greeting-service \
~/D0400/greeting-service
...output omitted...
[user@host D0400]$ cd greeting-service
[user@host greeting-service]$
```



Important

Windows users must replace the preceding `cp` command in PowerShell as follows:

```
PS C:\Users\user\D0400> Copy-Item -Path D0400-apps\greeting-service `
>> -Destination ~\D0400\greeting-service -Recurse
```

- 6.2. Create an empty public repository in GitHub named **do400-greeting-service** to host the source code of the greeting service.
- 6.3. Initialize a local repository in the `~/D0400/greeting-service` folder and add the **origin** remote.

```
[user@host greeting-service]$ git init .
Initialized empty Git repository in ...output omitted.../D0400/greeting-
service/.git/
[user@host greeting-service]$ git remote \
add origin https://github.com/YOUR_GITHUB_USER/do400-greeting-service.git
```

6.4. Commit the current version of the code and push it to the origin repository.

```
[user@host greeting-service]$ git add -A
[user@host greeting-service]$ git commit -m "Initial commit"
[master (root-commit) e453e03] Initial commit
...output omitted...
[user@host greeting-service]$ git branch -M main
[user@host greeting-service]$ git push origin main
...output omitted...
To https://github.com/your_github_user/do400-greeting-service.git
 * [new branch]      main -> main
```

If prompted, enter your GitHub user name, and personal access token.

- ▶ 7. Create the `greeting-service` application in OpenShift. Your continuous integration pipeline will redeploy this application after each change.

7.1. Create the `greeting-service` application in OpenShift:

```
[user@host greeting-service]$ oc new-app --name greeting-service \
https://github.com/YOUR_GITHUB_USER/do400-greeting-service \
--strategy=docker
...output omitted...
--> Success
    Build scheduled, use 'oc logs -f bc/greeting-service' to track its progress.
    Application is not exposed. You can expose services to the outside world by
    executing one or more of the commands below:
        'oc expose svc/greeting-service'
    Run 'oc status' to view your app.
```

Note the `strategy` option set to `docker`. This builds the application image by using the `Dockerfile` in the `do400-greeting-service` repository.

7.2. Wait for the application to deploy. Launch the `oc logs` command to watch the progress of the build.

```
[user@host greeting-service]$ oc logs -f bc/greeting-service
...output omitted...
```

Note the `-f` option sets it to follow updates as they occur, and the `bc/greeting-service` parameter instructs the command to watch the `greeting-service` BuildConfig.

7.3. Expose the `greeting-service` application to create a route to the application.

```
[user@host greeting-service]$ oc expose svc/greeting-service
route.route.openshift.io/greeting-service exposed
```

- 7.4. Get the application route.

```
[user@host greeting-service]$ oc get route greeting-service \
-o jsonpath="{ 'http://' }{ .spec.host }{ '\n' }"
http://greeting-service-your-user-greetings.apps.cluster.example.com
```

- 7.5. Open a new tab in the browser, and navigate to the URL that you just retrieved. Verify that the application response is **Hello guest!**.

► 8. Create the pipeline for the `greeting-service` application in Jenkins Blue Ocean.

- 8.1. In your web browser, navigate to the Jenkins main page and click **Open Blue Ocean** in the left navigation pane.
- 8.2. In the Blue Ocean main page, click **New Pipeline** in the upper right.
- 8.3. In the **Where do you store your code?** step, select **GitHub**. Jenkins uses the token you specified for the `greeting-console` application, so you do not need to specify the token again.
- 8.4. In the **Which organization does the repository belong to?** step, select your GitHub user name.
- 8.5. In the **Choose a repository** step, select the `do400-greeting-service` repository from the list and click **Create Pipeline**. After creating the pipeline, Jenkins triggers the first pipeline run. The pipeline already includes a **Check Style** stage and a **Test** stage. The **Test** stage should fail due to the following error:

```
...output omitted...
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
+ actual - expected

+ 'Ciao Guy!'
- 'Hello Guy!'
...output omitted...
```

- 8.6. Set up the webhook of the `do400-greeting-service` repository. Navigate to the Jenkins homepage, click **Manage Jenkins** in the left pane, and then click **Configure System** in the right pane.
- 8.7. Scroll down to the **GitHub** area, click **Advanced**, and then click **Re-register hooks for all jobs**.
- 8.8. Navigate to the webhooks page of the `do400-greeting-service` repository located at `https://github.com/YOUR_GITHUB_USER/do400-greeting-service/settings/hooks`. Verify that the repository has a webhook with a green tick.
- 9. Fix the code to make the tests pass. Modify the pipeline to continuously deploy the application.
- 9.1. Open the `~/DO400/greeting-service/Jenkinsfile` file and add a new **Deploy** stage. This stage deploys the `greeting-service` to OpenShift.

```
stage('Deploy') {
  steps {
    sh '''
      oc project RHT_OCP4_DEV_USER-greetings
      oc start-build greeting-service --follow --wait
    '''
  }
}
```

Replace *RHT_OCP4_DEV_USER* with your lab user name. Note how this starts the *greeting-service* that you configured previously.

9.2. Install the application dependencies and run the tests.

```
[user@host greeting-service]$ npm ci
...output omitted...
[user@host greeting-service]$ npm test
...output omitted...
1) greet
   greets in italian:

      AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
+ actual - expected

+ 'Ciao Guy!'
- 'Hello Guy'
...output omitted...
```

9.3. Open the `~/D0400/greeting-service/greet.js` file and modify the code to make the test pass. Change the greeting message to say *Ciao*.

```
module.exports = function greet(name) {
  return `Ciao ${name || "guest"}!`;
}
```

Save the file.

9.4. Run the tests again and verify they pass.

```
[user@host greeting-service]$ npm test
...output omitted...
greet
  # greets in italian

1 passing (2ms)
```

9.5. Commit and push the changes.

```
[user@host greeting-service]$ git commit -a -m "Fix greeting and add Deploy stage"
[main a74e3a7] Fix greeting and add Deploy stage
 2 files changed, 9 insertions(+), 1 deletion(-)
[user@host greeting-service]$ git push origin main
To https://github.com/your_github_user/do400-greeting-service.git
...output omitted... main -> main
```

- 9.6. In your web browser, navigate to the Jenkins Blue Ocean main page. Select the **do400-greeting-service** project and verify that the pipeline executes the **Deploy** stage and succeeds. The logs from the **Deploy** stage should show that the image has been pushed.

```
...output omitted...
oc start-build greeting-service --follow --wait
...output omitted...
Push successful
```

- 9.7. Return to the application tab in your browser. Refresh the page and verify that the application response is **Ciao guest**.

- **10.** Delete the **greetings** project and navigate back to your workspace folder.

```
[user@host greeting-service]$ oc delete project RHT_OCP4_DEV_USER-greetings
[user@host greeting-service]$ cd ..
[user@host D0400]$
```

This concludes the guided exercise.

Deploying Applications to Various Environments

Objectives

After completing this section, you should be able to use checkpoints to deploy to various environments.

Describing Deployment Environments

A deployment environment is a system in which an application is deployed and executed. You can use tools such as Jenkins and Red Hat OpenShift to deploy and execute your applications in those systems. Having multiple environments allows you to rigorously test your application before it is available to your users.

The following non-comprehensive list shows the most common classification of deployment environments.

- **Local:** environment where developers experiment the application. This environment may also be called SANDBOX, regarding where the developers do their experiments.
- **Development:** environment where developers test and experiment with an application.
- **Testing:** environment where the quality control team perform tests to verify the correct operation of an application.
- **Staging:** environment that mirrors the production environment, to test an application before the changes are applied to the production environment. This environment may also be called PRE-PROD. In some special cases, if there is no User Acceptance Testing (UAT) environment, then teams occasionally run user acceptance tests in this environment.
- **Production:** environment used by the end users of an application.

Storing Build Artifacts

The purpose of a pipeline is to build and deliver new versions of your application to your users. The way you distribute the build artifacts is up to your project requirements. The two most common ways of distributing them is by providing an artifact file or inside a container image. With Jenkins, you can store the artifacts in the workspace, or push them to an external system.

Using Jenkins

With the `archiveArtifacts` step, you can store build artifacts, to download later. Jenkins stores the artifacts per build, and keeps them as long as you keep the build itself.

The basic usage of the `archiveArtifacts` step is to provide a string parameter with the files you want to archive. You can use wildcards in the string parameter. The following example shows the usage of the `archiveArtifacts` step.

```
pipeline {
  agent any
  stages {
    ...output omitted...
```

```

stage('Package') {
    steps {
        sh './mvnw package -DskipTests'
        archiveArtifacts 'target/*.jar'
    }
}
}
}

```

You can access the latest artifacts from the pipeline details page or the build details page for older artifacts.

Using an Image Registry

As seen in *Building Images and Deploying to OpenShift*, you can use container images to distribute and deploy your applications. With Quarkus, you can build container images and push them to a registry.

The following Quarkus extensions allow you to create container images:

- **Jib**: creates container images without external dependencies such as the Docker or Source-to-Image binaries.
- **Docker**: uses the Docker binary to create the container images.
- **Source-to-Image**: uses the source-to-image (S2I) binary to create container images inside a Red Hat OpenShift cluster.

The following command builds and pushes a container image to Quay.io with Quarkus.

```

[user@host ~]$ ./mvnw package \
-Dquarkus.jib.base-jvm-image=quay.io/redhattrain \
-ing/do400-java-alpine-openjdk11-jre:latest \ ❶
-Dquarkus.container-image.build=true \ ❷
-Dquarkus.container-image.registry=quay.io \ ❸
-Dquarkus.container-image.group=YOUR_QUAY_USER_OR_GROUP \ ❹
-Dquarkus.container-image.name=IMAGE_NAME \ ❺
-Dquarkus.container-image.username=QUAY_USER \ ❻
-Dquarkus.container-image.password=QUAY_PASSWORD \ ❼
-Dquarkus.container-image.push=true ❽

```

- ❶ The base image to use
- ❷ Whether to create an image
- ❸ The container registry to use
- ❹ The user or group in the container registry
- ❺ The container image name
- ❻ The image registry user
- ❼ The image registry password
- ❽ Whether to push the image to the repository

Constructing Dynamic Pipelines

Pipelines should not be rigid scripts, they should be able to adapt to a multitude of scenarios.

For example, you might need to use a checkpoint, and request confirmation from a user to deploy to an environment. In this case, use the `input` directive in your pipeline scripts. This instructs Jenkins to pause the stage during execution and prompt for input. When you approve the input, the stage execution continues, otherwise Jenkins aborts the execution.

Other options to create more dynamic and secure pipelines include the use of environment variables, credentials, and input prompts, among others.

Environment Variables

An environment variable is a dynamically named value that modifies pipeline behavior. Use the `environment` directive in pipeline scripts to define environment variables. In Jenkins pipelines, you can set environment variables that are valid globally, or only in a specific stage.

Global environment variables

The environment variables defined in the root of the `pipeline` block are available anywhere within the pipeline script. The following example defines a global environment variable.

```
pipeline {
  agent any
  environment { PERSON = 'Joel' }
  stages {
    stage('Hello') {
      steps {
        echo "Hello ${PERSON}"
      }
    }
  }
}
```

Per-stage environment variables

The environment variables defined inside a `stage` directive are only available within that stage. The following example shows the definition and usage of an environment variable inside a stage.

```
pipeline {
  agent any
  stages {
    stage('Hello') {
      environment { PERSON = 'Aykut' } ❶
      steps {
        echo "Hello ${PERSON}" ❷
      }
    }
    stage('Bye') {
      steps {
        echo "Bye ${PERSON}" ❸
      }
    }
  }
}
```

- ❶ Creates an environment variable at the stage level
- ❷ References the environment variable in the stage in which it was defined
- ❸ **Incorrect**, as the `PERSON` environment variable is not available in this stage, and the pipeline execution fails

Credentials

Most pipelines require secrets to authenticate with external resources such as repositories, registries, and servers. With Jenkins, you can store encrypted secrets and access them in your pipeline scripts. The most common types of credentials stored in Jenkins are the following: *secret text*, *username and password*, and *secret file*.

Navigate to the Jenkins homepage, click **Manage Jenkins** in the left pane, and then click **Manage Credentials** in the right pane to manage stored credentials.

To use credentials in your pipeline scripts, you can use the `credentials` helper inside the `environment` directive. This helper only works with the *secret text*, *username and password*, and *secret file* types. It accesses the credentials by their identifier and returns the value.

The following example shows the retrieval and use of a credential stored in Jenkins.

```
...output omitted...
stage('Push Image to Registry') {
    environment { QUAY = credentials('MY_QUAY_USER') } ❶
    steps {
        ...output omitted...
        sh '''
            ./mvnw package \
            -Dquarkus.jib.base-jvm-image=quay.io/redhattraining/do400-java-alpine-
            openjdk11-jre:latest \
            -Dquarkus.container-image.build=true \
            -Dquarkus.container-image.registry=quay.io \
            -Dquarkus.container-image.group=YOUR_QUAY_USER_OR_GROUP \
            -Dquarkus.container-image.name=YOUR_IMAGE_NAME \
            -Dquarkus.container-image.username=$QUAY_USR \ ❷
            -Dquarkus.container-image.password=$QUAY_PSW \ ❸
            -Dquarkus.container-image.push=true
        '''
    }
}
...output omitted...
```

- ❶ Retrieves the credential with identifier `MY_QUAY_USER`, and creates an environment variable named `QUAY` with the value of the credential. The credential is a *username and password* type, so the helper also creates two auxiliary environment variables by concatenating `_USR` and `_PSW` to the variable name. The `QUAY_USR` variable contains the username, and the `QUAY_PSW` variable contains the password.
- ❷ Uses the auxiliary variable that contains the username stored in the credential.
- ❸ Uses the auxiliary variable that contains the password stored in the credential.



References

Source-to-Image (S2I)

<https://github.com/openshift/source-to-image>

Quarkus - Container Images

<https://quarkus.io/guides/container-image>

Jenkins Core

<https://www.jenkins.io/doc/pipeline/steps/core/>

Using Credentials in Jenkins

<https://www.jenkins.io/doc/book/using/using-credentials/>

► Guided Exercise

Deploying Applications to Various Environments

In this exercise you will create a pipeline that deploys an application to different environments.

The solution files are provided at the `shopping-cart-v3` folder in the `solutions` branch of the `DO400-apps` repository. Notice that you might need to replace some strings in the `Jenkinsfile` file.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to create multiple environments in OpenShift, archive artifacts, and deploy the same container image to different environments.

Before You Begin

To perform this exercise, ensure you have:

- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- Maven and a JDK
- The OpenShift CLI
- A Quay.io account



Important

Try to keep your course work organized by creating a workspace folder, such as `~/DO400`. Start the guided exercise from this folder.

Instructions

- 1. Copy the Shopping Cart application source from the `solutions` branch of your `DO400-apps` fork.
 - 1.1. Open a command-line terminal and checkout the `solutions` branch in your `DO400-apps` fork.

```
[user@host D0400]$ cd ~/D0400/D0400-apps
[user@host D0400-apps]$ git checkout solutions
Branch 'solutions' set up to track remote branch 'solutions' from 'origin'.
Switched to a new branch 'solutions'
```

- 1.2. Copy the content of the `shopping-cart` folder to the `deploying-envs` folder.

```
[user@host D0400-apps]$ cp -Rv shopping-cart/ \
~/D0400/deploying-envs
...output omitted...
```



Important

Windows users must also run the preceding command but in PowerShell as follows:

```
PS C:\Users\user\D0400\D0400-apps> Copy-Item -Path shopping-cart `
>> -Destination ~\D0400\deploying-envs -Recurse
```

- 1.3. Checkout the `main` branch and navigate to the new directory.

```
[user@host D0400-apps]$ git checkout main
Switched to branch 'main'
...output omitted...
[user@host D0400-apps]$ cd ~/D0400/deploying-envs
[user@host deploying-envs]$
```

- ▶ 2. Create a public, empty Quay Container Image repository called `do400-deploying-environments`.
 - 2.1. Open a web browser and navigate to <https://quay.io>.
 - 2.2. Sign in with your Quay credentials. Create an account if you do not already have one.
 - 2.3. Click **Create New Repository** in the upper right of the window.
 - 2.4. Select the **Container Image Repository** in the repository type list. In the case where you are part of an organization, make sure that the repository is created under your Quay user. Fill in the **Repository name** field with the `do400-deploying-environments` value.
 - 2.5. Click **Public** in the **Repository Visibility** list of options. Do not change any other field.
 - 2.6. Click **Create Public Repository**.
- ▶ 3. Create a public GitHub repository to store the source code of the Shopping Cart application.
 - 3.1. Open a new tab in the browser and navigate to <https://github.com>.
 - 3.2. Sign in with your GitHub credentials. Create an account if you do not already have one.

- 3.3. Click + → **New repository** in the upper right of the window.
- 3.4. Fill in the **Repository name** field with the `do400-deploying-environments` value. Do not change any other field. Click **Create repository**.
- 3.5. Return to the command-line terminal, initialize a local repository in the `~/D0400/deploying-envs` folder, and add the remote origin.

```
[user@host deploying-envs]$ git init .
Initialized empty Git repository in ...output omitted.../D0400/deploying-
envs/.git/
[user@host deploying-envs]$ git remote \
add origin https://github.com/YOUR_GITHUB_USER/do400-deploying-environments.git
```

Notice that you must replace the `YOUR_GITHUB_USER` string with your GitHub user.

- 3.6. Commit the current version of the code, set `main` as the default branch, and push it to the `origin` repository.

```
[user@host deploying-envs]$ git add -A
```



Important

Windows users must also execute the following command after the `git add` command:

```
PS C:\Users\user\D0400\deploying-envs> git update-index --chmod=+x mvnw
```

```
[user@host deploying-envs]$ git commit -m "Initial commit"
[master (root-commit) 85b6235] Initial commit
...output omitted...
[user@host deploying-envs]$ git branch -M main
[user@host deploying-envs]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-deploying-environments.git
* [new branch]      HEAD -> main
```

If prompted, enter your GitHub user name, and personal access token.

- ▶ 4. Create a scripted pipeline composed of different stages executed sequentially. The requirements for the pipeline are the following:
 - The pipeline uses a Maven node.
 - The pipeline includes a Test stage.
 - The pipeline includes a Package stage that creates an uber-JAR file, and stores it in Jenkins.
 - The Package stage can skip the tests.
- 4.1. Create a file named `Jenkinsfile` in the root of the application folder to store the pipeline.

In the **Package** stage, use the `-DskipTests` property to skip the tests execution, and the `-Dquarkus.package.type=uber-jar` property to create a package with the application and all the required dependencies.

The file contents should look like the following:

```
pipeline {
  agent {
    node {
      label 'maven'
    }
  }
  stages {
    stage('Tests') {
      steps {
        sh './mvnw clean test'
      }
    }
    stage('Package') {
      steps {
        sh '''
          ./mvnw package -DskipTests \
          -Dquarkus.package.type=uber-jar
        '''
        archiveArtifacts 'target/*.jar'
      }
    }
  }
}
```

4.2. Stage and commit the `Jenkinsfile` file to the local repository.

```
[user@host deploying-envs]$ git add Jenkinsfile
[user@host deploying-envs]$ git commit -m "Added Jenkins integration"
[main eb6ad6c] Added Jenkins integration
1 file changed, 20 insertions(+)
create mode 100644 Jenkinsfile
```

4.3. Push the local commits to the remote repository with the `git push` command.

```
[user@host deploying-envs]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-deploying-environments.git
97c6b69..0ff54e8 HEAD -> main
```

- ▶ 5. Create a Jenkins project to run the pipeline defined in the `do400-deploying-environments` repository.
 - 5.1. Open a new tab in the browser and navigate to the URL of the Jenkins server, which you installed in *Guided Exercise: Configuring a Developer Environment*.
 - 5.2. The Jenkins server is bound to your OpenShift account. If prompted, log into Jenkins with your OpenShift credentials.

- 5.3. In the left pane click **New Item**.
- 5.4. Assign the name `do400-deploying-environments` to the project, then click **Pipeline**, and finally click **OK**.
- 5.5. Select the **Discard old builds** check box, and then type 3 in the **Max # of builds to keep** field.
- 5.6. Click the **Pipeline** tab. In the **Pipeline** area, from the **Definition** list, select **Pipeline script from SCM**.
- 5.7. From the **SCM** list select **Git**. In the **Repository URL** type the URL of the GitHub repository created in the first steps.
The URL of the repository looks like `https://github.com/YOUR_GITHUB_USER/do400-deploying-environments`.
- 5.8. In the **Branch Specifier** field, type `*/main`.
- 5.9. Click **Save** to save the pipeline and go to the pipeline details page.

► **6.** Run the pipeline to check that the Jenkins integration was successful.

- 6.1. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful.
- 6.2. Reload the pipeline details page to see the **Last Successful Artifacts** links in the pipeline details page. You can also access a specific build and download its generated artifacts.
- 6.3. Click `shopping-cart-1.0-SNAPSHOT-runner.jar` and save the latest packaged application in your workspace folder.
- 6.4. Open a new command-line terminal and navigate to your workspace folder.

```
[user@host deploying-envs]$ cd ~/D0400
[user@host D0400]$
```

- 6.5. Start the shopping cart application. Use the application packaged by Jenkins.

```
[user@host D0400]$ java -jar shopping-cart-1.0-SNAPSHOT-runner.jar
...output omitted...
...output omitted... Listening on: http://localhost:8080
...output omitted...
```

- 6.6. Open a new tab in the browser and navigate to `http://localhost:8080/cart`. Notice that the Quarkus application returns a JSON response composed of the `products` key with an empty list as value, and the `totalItems` key with 0 as value.
- 6.7. Close the browser tab, return to the command-line terminal that has the application running. Press **CTRL + c** to stop the running application and close the command-line terminal.

► **7.** Add a stage to the pipeline to generate a container image and push it to Quay.

- 7.1. Return to the Jenkins tab in your browser, and then click the **Jenkins** logo in the upper left to navigate to the Jenkins homepage.

- 7.2. Click **Manage Jenkins** in the left pane, and then click **Manage Credentials** in the right pane.
- 7.3. In the **Stores scoped to Jenkins** section of the right pane, click **Jenkins**.
- 7.4. Click **Global credentials (unrestricted)** in the right pane, and then click **Add Credentials** in the left pane.
- 7.5. From the **Kind** list, select **Username with password**. In the **Username** field, type your Quay username. In the **Password** field, type your Quay password. In the **ID** field, type **QUAY_USER**, and then click **OK**.
- 7.6. Open the **Jenkinsfile** file and add a stage named **Build Image** to execute after the **Package** stage. The stage generates a container image and pushes it to Quay by using the credentials stored in Jenkins.

The content of the file should look as follows:

```
...output omitted...
stage('Build Image') {
    environment { QUAY = credentials('QUAY_USER') } ❶
    steps {
        sh '''
            ./mvnw quarkus:add-extension \
            -Dextensions="kubernetes,container-image-jib" ❷
            ...
        sh '''
            ./mvnw package -DskipTests \
            -Dquarkus.jib.base-jvm-image=quay.io/redhattraining/do400-java-alpine-
            openjdk11-jre:latest \
            -Dquarkus.container-image.build=true \ ❸
            -Dquarkus.container-image.registry=quay.io \ ❹
            -Dquarkus.container-image.group=$QUAY_USR \ ❺
            -Dquarkus.container-image.name=do400-deploying-environments \ ❻
            -Dquarkus.container-image.username=$QUAY_USR \ ❼
            -Dquarkus.container-image.password="$QUAY_PSW" \ ❽
            -Dquarkus.container-image.push=true ❾
            ...
        }
    }
}
...output omitted...
```

- ❶ The `credentials` helper method accesses Jenkins' credentials and stores the values in an environment variable. The helper also creates a `QUAY_USER_USR` environment variable with the user name and a `QUAY_USER_PSW` environment variable with the password.
- ❷ The `container-image-jib` extension allows the generation of container images without dedicated client side tooling like Docker.
- ❸ Option that enables the generation of the container image in Quarkus.
- ❹ Container image registry that stores the container images.
- ❺ Group or user that the container image will be part of.
- ❻ Container image name.

- 7. User for the image registry.
- 8. Password for the image registry.
- 9. Option that enables the push of the container image to the specified image registry.

**Note**

You can define all the packaging options in the `pom.xml` file instead of passing them as parameters to the Maven goal.

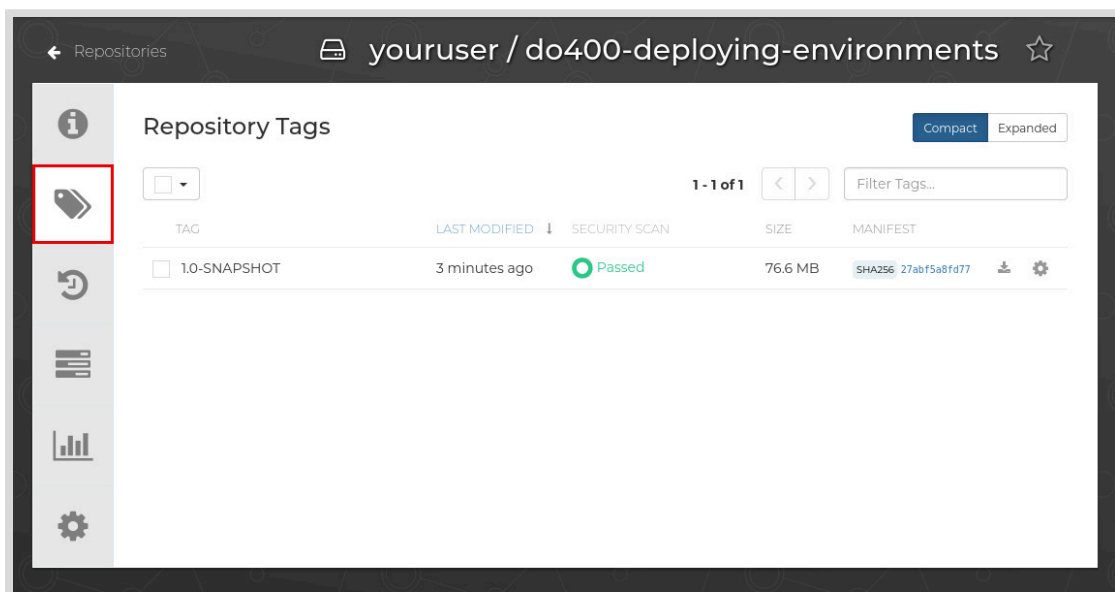
- 7.7. Stage and commit the `Jenkinsfile` file to the local repository.

```
[user@host deploying-envs]$ git add Jenkinsfile
[user@host deploying-envs]$ git commit -m "Added Build Image stage"
[main eb6ad6c] Added Build Image stage
1 file changed, 18 insertions(+)
```

- 7.8. Push the local commits to the remote repository with the `git push` command.

```
[user@host deploying-envs]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-deploying-environments.git
97c6b69..0ff54e8 HEAD -> main
```

- 7.9. Return to the Jenkins tab in your browser, click the **Jenkins** logo in the upper left, and then click **do400-deploying-environments** in the right pane. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful.
- 7.10. Open the Quay tab of your browser, refresh the page, and navigate to the **Tags** section of the container image repository. Notice that a new image was pushed and assigned the **1.0-SNAPSHOT** tag.



**Note**

You can run the pushed image with **podman** in your local environment or in a container platform such as Red Hat OpenShift.

► **8.** Create two environments in OpenShift to deploy the application.

- 8.1. Return to the command-line terminal and log in to OpenShift with the OpenShift CLI. To log in, use the credentials provided in the **Lab Environment** tab of the course's online learning website, after you provision your online lab environment.

```
[user@host deploying-envs]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
```

- 8.2. Create a new OpenShift project to host the production environment of the Shopping Cart application.

```
[user@host deploying-envs]$ oc new-project \
RHT_OCP4_DEV_USER-shopping-cart-production
Now using project "youruser-shopping-cart-production" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

Notice that you must replace the **RHT_OCP4_DEV_USER** string with your OpenShift developer user.

- 8.3. Deploy the Shopping Cart application manually to the production environment.

```
[user@host deploying-envs]$ oc process \
-n RHT_OCP4_DEV_USER-shopping-cart-production \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_ENVIRONMENT=production \
| oc apply -n RHT_OCP4_DEV_USER-shopping-cart-production -f -
deployment.apps/shopping-cart-production created
service/shopping-cart-production created
route.route.openshift.io/shopping-cart-production created
```

Notice that you must replace the **RHT_OCP4_DEV_USER** string with your OpenShift developer user, and the **YOUR_QUAY_USER** string with your Quay username.

- 8.4. Add the **edit** role to the Jenkins account in the **RHT_OCP4_DEV_USER-shopping-cart-production** project.

```
[user@host deploying-envs]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"
```

Notice that you must replace the **RHT_OCP4_DEV_USER** string with your OpenShift developer user.

- 8.5. Create a new OpenShift project to host the staging environment of the Shopping Cart application.

```
[user@host deploying-envs]$ oc new-project RHT_OCP4_DEV_USER-shopping-cart-stage
Now using project "youruser-shopping-cart-stage" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

Notice that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user.

- 8.6. Deploy the Shopping Cart application manually to the staging environment.

```
[user@host deploying-envs]$ oc process \
-n RHT_OCP4_DEV_USER-shopping-cart-stage \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_ENVIRONMENT=stage \
| oc apply -n RHT_OCP4_DEV_USER-shopping-cart-stage -f -
deployment.apps/shopping-cart-stage created
service/shopping-cart-stage created
route.route.openshift.io/shopping-cart-stage created
```

Notice that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user, and the `YOUR_QUAY_USER` string with your Quay username.

- 8.7. Add the edit role to the Jenkins account in the `RHT_OCP4_DEV_USER-shopping-cart-stage` project.

```
[user@host deploying-envs]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"
```

Notice that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user.

► 9. Add a deployment stage for the staging environment.

- 9.1. Open the `Jenkinsfile` file and add the following environment variables:

- **RHT_OCP4_DEV_USER**: to store your OpenShift developer user.
- **DEPLOYMENT_STAGE**: to store the name of the `Deployment` resource for the staging environment.
- **DEPLOYMENT_PRODUCTION**: to store the name of the `Deployment` resource for the production environment.

The code should look like the following:

```
pipeline {
  agent {
    node {
      label 'maven'
```

```

    }
  }
  environment {
    RHT_OCP4_DEV_USER = 'YOUR_OCP_DEV_USER'
    DEPLOYMENT_STAGE = 'shopping-cart-stage'
    DEPLOYMENT_PRODUCTION = 'shopping-cart-production'
  }
  stages {
...output omitted...

```

Notice that you must update the `RHT_OCP4_DEV_USER` value. Use the user name provided in the `Lab Environment` tab of the course's online learning website, after you provision your online lab environment.

- 9.2. Modify the `./mvnw package` command in the `Build Image` stage to tag the application image with the Jenkins build number. In this way, each pipeline build will push a different image tag.

Add the `quarkus.container-image.tag` parameter used to generate the image tag with the Jenkins build number. The image tag should follow the `build-${BUILD-NUMBER}` format.

The updated command should look like the following:

```

...output omitted...
./mvnw package -DskipTests \
  -Dquarkus.jib.base-jvm-image=quay.io/redhattraining/do400-java-alpine-openjdk11-jre:latest \
  -Dquarkus.container-image.build=true \
  -Dquarkus.container-image.registry=quay.io \
  -Dquarkus.container-image.group=$QUAY_USR \
  -Dquarkus.container-image.name=do400-deploying-environments \
  -Dquarkus.container-image.username=$QUAY_USR \
  -Dquarkus.container-image.password='$QUAY_PSW' \
  -Dquarkus.container-image.tag=build-${BUILD_NUMBER} \
  -Dquarkus.container-image.push=true
...output omitted...

```

You do not need to initialize the `BUILD_NUMBER` variable because Jenkins provides this environment variable.

- 9.3. Add a deployment stage named `Deploy - Stage` after the `Build Image` stage. In this stage, Jenkins changes the Deployment to use the `build-${BUILD_NUMBER}` image tag. This rolls out the new image tag.

The code should look like the following:

```

...output omitted...
stage('Deploy - Stage') {
  environment {
    APP_NAMESPACE = "${RHT_OCP4_DEV_USER}-shopping-cart-stage"
    QUAY = credentials('QUAY_USER')
  }
  steps {
    sh """
      oc set image \
      deployment ${DEPLOYMENT_STAGE} \

```

```

        shopping-cart-stage=quay.io/${QUAY_USR}/do400-deploying-
environments:build-${BUILD_NUMBER} \
        -n ${APP_NAMESPACE} --record
    ""
}
}
...output omitted...

```

**Note**

The `--record` flag records the executed command in the Deployment resource annotations. This is useful to later inspect the causes of Deployment rollouts, for example, by using the `oc rollout history` command.

The preceding code uses the `oc set image` command to update the deployment with the new image tag. The update triggers a new rollout.

- 9.4. Stage and commit the `Jenkinsfile` file to the local repository.

```

[user@host deploying-envs]$ git add Jenkinsfile
[user@host deploying-envs]$ git commit -m "Added staging deployment"
[main eb6ad6c] Added staging deployment
1 file changed, 11 insertions(+)

```

- 9.5. Push the local commits to the remote repository with the `git push` command.

```

[user@host deploying-envs]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-deploying-environments.git
97c6b69..0ff54e8    HEAD -> main

```

- 9.6. Return to the Jenkins tab in your browser. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful.
- 9.7. Return to the command-line terminal. Validate that the deployment was successfully rolled out with the `oc rollout status` command.

```

[user@host deploying-envs]$ oc rollout status deployment shopping-cart-stage \
-n RHT_OCP4_DEV_USER-shopping-cart-stage
deployment "shopping-cart-stage" successfully rolled out

```

Note that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user.

- 9.8. Check that the `oc rollout history` command lists the new rollout. The new rollout should indicate the command that causes the change. In this case, the cause is the command that Jenkins executed, including the latest image tag.

```
[user@host deploying-envs]$ oc rollout history deployment shopping-cart-stage \
-n RHT_OCP4_DEV_USER-shopping-cart-stage
REVISION  CHANGE-CAUSE
1          <none>
2          oc set image deployment shopping-cart-stage ...output
omitted...:build-3 ...output omitted...
```

Notice that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user.

► 10. Add a deployment stage to push to the production environment.

- 10.1. Open the `Jenkinsfile` file, and add a deployment stage named `Deploy - Production` after the `Deploy - Stage` stage. Request for confirmation before the stage is executed.

The code should look like the following:

```
...output omitted...
stage('Deploy - Production') {
    environment {
        APP_NAMESPACE = "${RHT_OCP4_DEV_USER}-shopping-cart-production"
        QUAY = credentials('QUAY_USER')
    }
    input { message 'Deploy to production?' }
    steps {
        sh """
            oc set image \
                deployment ${DEPLOYMENT_PRODUCTION} \
                shopping-cart-production=quay.io/${QUAY_USR}/do400-deploying-
environments:build-${BUILD_NUMBER} \
                -n ${APP_NAMESPACE} --record
        """
    }
}
...output omitted...
```

The deployment stage rolls out the Deployment in the production environment, only when a Jenkins user approves the execution of the stage.

- 10.2. Stage and commit the `Jenkinsfile` file to the local repository.

```
[user@host deploying-envs]$ git add Jenkinsfile
[user@host deploying-envs]$ git commit -m "Added production deployment"
[main eb6ad6c] Added production deployment
1 file changed, 7 insertions(+)
```

- 10.3. Push the local commits to the remote repository with the `git push` command.

```
[user@host deploying-envs]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-deploying-environments.git
97c6b69..0ff54e8 HEAD -> main
```

- 10.4. Return to the Jenkins tab in your browser. In the left pane, click **Build Now** to schedule a pipeline run. Wait until the pipeline pauses in the **Deploy - Production** stage.
- 10.5. Return to the command-line terminal. Check the rollout history of the production environment with the `oc rollout history` command. Only one rollout exists.

```
[user@host deploying-envs]$ oc rollout history \
deployment shopping-cart-production \
-n RHT_OCP4_DEV_USER-shopping-cart-production
REVISION  CHANGE-CAUSE
1          <none>
```

- 10.6. Return to the Jenkins tab in your browser. Hover with your mouse over the running **Deploy - Production** stage, and then on the pop-up menu, click **Abort** to stop the pipeline execution.
- 10.7. Return to the command-line terminal. Check the rollout history of the production environment with the `oc rollout history` command.

```
[user@host deploying-envs]$ oc rollout history \
deployment shopping-cart-production \
-n RHT_OCP4_DEV_USER-shopping-cart-production
REVISION  CHANGE-CAUSE
1          <none>
```

Notice that the production deployment was not executed.

- 10.8. In the left pane, click **Build Now** to schedule a pipeline run. Wait until the pipeline pauses in the **Deploy - Production** stage.
- 10.9. Hover with your mouse over the running **Deploy** stage, and then on the pop-up menu, click **Proceed** to continue the pipeline execution.
- 10.10. Wait for the build to finish, and check that the build was successful.
- 10.11. Return to the command-line terminal. Verify that Jenkins rolled out the Deployment of the production environment.

```
[user@host deploying-envs]$ oc rollout history \
deployment shopping-cart-production \
-n RHT_OCP4_DEV_USER-shopping-cart-production
REVISION  CHANGE-CAUSE
1          <none>
2          oc set image deployment shopping-cart-production ...output omitted...
```

► 11. Clean up.

- 11.1. Delete the OpenShift projects and navigate back to your workspace folder.


```
[user@host deploying-envs]$ oc delete project \
RHT_OCP4_DEV_USER-shopping-cart-production
project.project.openshift.io "your-user-shopping-cart-production" deleted
[user@host deploying-envs]$ oc delete project \
RHT_OCP4_DEV_USER-shopping-cart-stage
project.project.openshift.io "your-user-shopping-cart-stage" deleted
[user@host deploying-envs]$ cd ..
[user@host DO400]$
```

- 11.2. Return to the Jenkins tab in your browser, and click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.

This concludes the guided exercise.

Implementing Release Strategies

Objectives

After completing this section, you should be able to release applications by using various strategies to mitigate downtime and risk.

Defining Deployment Strategies

The process of deploying changes or new versions of your application is an important stage of your CI/CD pipelines. It is the phase in which you deliver functional versions of your application to your users and therefore, you are adding value to your development process.

Introducing changes in your applications always carries risks, such as having downtimes during the deployment, introducing bugs, or reducing the application performance, among others. You can reduce or mitigate some risks by having testing and validation stages in your pipelines.

The downtime of your applications or services can imply losing business, disrupting other services that depend on yours, and service level agreement violations, among others. To reduce the downtime and minimize risks in deployments you should use a *deployment strategy*. A deployment strategy is a way of changing or upgrading an application, which minimizes the impact of those changes.

Defining Automatic Deploying Strategies in OpenShift

In Red Hat OpenShift, you use `Deployment` or `DeploymentConfig` objects to define deployments and deployment strategies. The **Rolling**, and the **Recreate** strategies are the main deployment strategies available in Red Hat OpenShift.

To set the Rolling or Recreate strategies, you must set the `.spec.strategy.type` property of the `Deployment` and `DeploymentConfig` objects. Note there is a slight difference between how `Deployment` and `DeploymentConfig` objects define the Rolling strategy.

- In `Deployment` objects, you can set the `.spec.strategy.type` property either to `RollingUpdate` or `Recreate`.
- In `DeploymentConfig` objects, you can set the `.spec.strategy.type` property either to `Rolling` or `Recreate`.

For example, the following snippet shows a `Deployment` object configured to use the *Recreate* strategy.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...output omitted...
spec:
  strategy:
    type: Recreate
  ...output omitted...
```

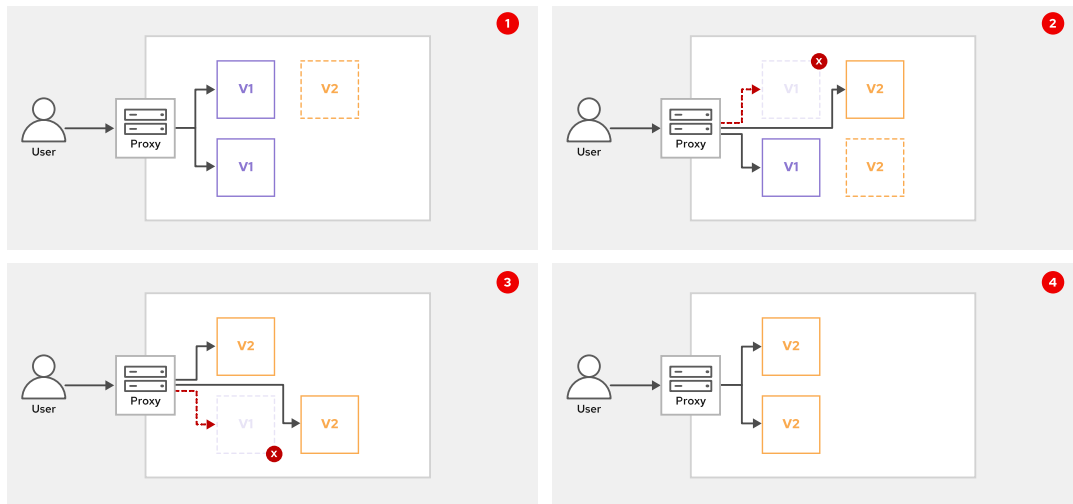
For advanced deploying strategies in Red Hat OpenShift you might require manual manipulation of routes.

Rolling Strategy

The **Rolling** strategy consists of slowly updating a version of an application. It replaces instances one after the other until there are no instances left to replace.

In this strategy, both versions of the application are going to run simultaneously, and it does not scale down instances of the previous version until the new one is ready. The major drawback of this strategy is that it requires compatibility between the versions involved in the deployment.

The following graphic shows the deployment of a new version of an application by using the Rolling strategy.



- 1 The application has a couple of instances running a version of the code that we want to update (v1). Red Hat OpenShift scales up a new instance with the new version of the application (v2). The new instance with version v2 is not ready, so the instances with version v1 are the only ones that fulfill customer requests.
- 2 The instance with v2 is ready and accepts customer requests. Red Hat OpenShift scales down an instance with version v1, and scales up a new instance with version v2. Both versions of the application fulfill customer requests.
- 3 The new instance with v2 is ready and accepts customer requests. Red Hat OpenShift scales down the remaining instance with version v1.
- 4 There are no instances left to replace. The application update was successful, and without downtime.

The Rolling strategy allows continuous deployment, and eliminates the downtime of your application during deployments. You can use this strategy if the different versions of your application can run at the same time.

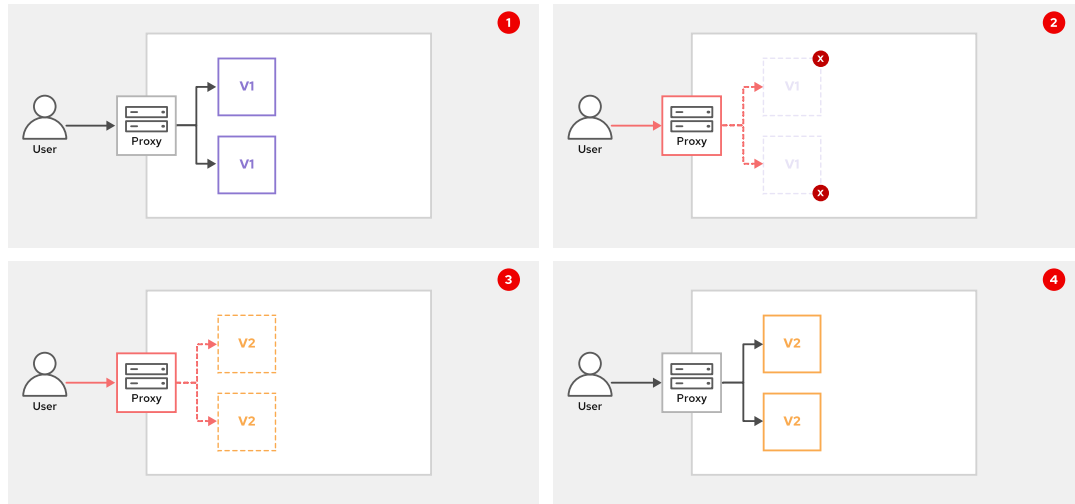
Note

The Rolling strategy is the default strategy used if you do not specify a strategy on the `Deployment` or `DeploymentConfig` objects.

Recreate Strategy

In this simple strategy, all the instances of an application are killed first, and then replaced with new ones. The major drawback of this strategy is that it causes a downtime in your services. For a period of time, there are no instances of your application to fulfill requests.

The following graphic shows the deployment of a new version of an application that uses the Recreate strategy.



- ❶ The application has a couple of instances running a version of the code that we want to update (v1).
- ❷ Red Hat OpenShift scales down the running instances to zero. This action causes a downtime in your application because there are no instances to fulfill requests.
- ❸ Red Hat OpenShift scales up new instances with a new version of the application (v2). The new instances are booting, so the downtime continues.
- ❹ The new instances finished booting, and are ready to fulfill requests. This is the last step of the Recreate strategy, and it resolves the outage of the application.

You can use this strategy when your application can not have different versions of the code running at the same time. You also might use it when you must execute data migrations or data transformations before the new code starts. This strategy is not recommended for applications that need high availability, for example, medical systems.

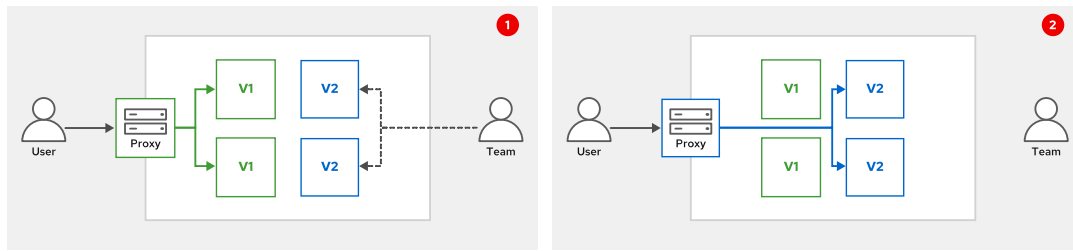
Defining Advanced Deployment Strategies

Red Hat OpenShift is a very flexible orchestration system, and you can adapt the deployment process to the requirements of your projects. You can implement advanced deploying strategies by manipulating routes, or by using additional components such as Red Hat OpenShift Service Mesh.

Blue-Green Strategy

In this strategy you have two versions of the application running simultaneously. Both versions are active, but only one of them is responsible for fulfilling your customer requests.

The following graphic shows the deployment of a new version of an application by using the Blue-Green strategy.



- 1 Two versions of the application run simultaneously. The current version (v1) processes the customer requests, while the team tests the new version (v2).
- 2 When the team successfully finishes testing the new version, they redirect the traffic to the new version. In the case of a problem, the team can quickly rollback to the previous version of the application.

The main advantage of this strategy is that you can quickly switch between different versions of your application. This strategy reduces the deployment downtime, but it increases the operational costs of your projects by duplicating the resources required to deploy the application.



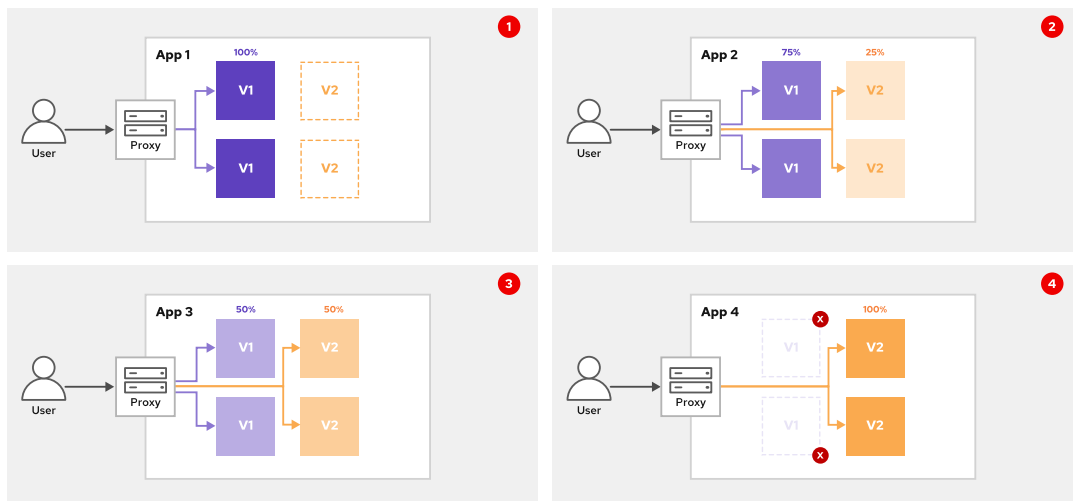
Note

If you are deploying a stateful application, then you must be sure that both versions have compatible data formats.

Canary Strategy

The **Canary** strategy is similar to the *Blue-Green* strategy, you have at least two versions of the application running simultaneously. Both versions are active, but only a small part of the traffic is redirected to the new version to test and validate the changes.

The following graphic shows the deployment of a new version of an application by using the Canary strategy.



- 1 Two versions of the application run simultaneously. The current version of the application receives all the traffic.
- 2 The team starts sending a small amount of traffic to the new version of the application.

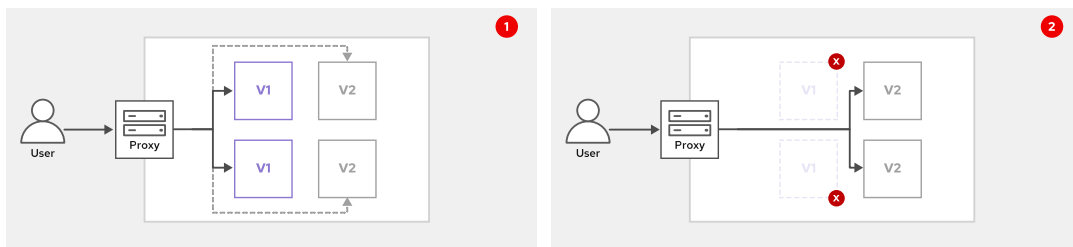
- 3 The team increases the traffic sent to the new version to measure and validate the changes added in the new version.
- 4 If the tests with live traffic are successful, then the team redirects all the traffic to the new version, and scales down the previous version.

The main advantage of this strategy is that you are performing tests with live traffic. The major drawback is that it increases the operational costs of your projects by duplicating the resources required to deploy the application. Similar to the Blue-Green strategy, if you are deploying a stateful application, then you must be sure that both versions have compatible data formats.

Shadow Strategy

The **Shadow** strategy is similar to the *Blue-Green* strategy, you have two versions of the application running simultaneously, but only one is receiving the real traffic. The second version receives a duplicate of the traffic sent to the other version. With this strategy, you can check the correctness of the new version, and compare the performance and results of both versions.

The following graphic shows the deployment of a new version of an application by using the Shadow strategy.



- 1 Two versions of the application run simultaneously, but only the v1 version receives the real traffic. The v2 version receives a duplicate of the traffic sent to the v1 version.
- 2 When the team validates the correctness of the v2 version, they redirect the traffic to the new version, and scale down the previous version.

This strategy is more complex to set up than the others. You can implement this strategy at the application level, writing the necessary code to send the duplicated traffic between versions. Another option is to implement the shadowing at the infrastructure layer, with the help of products such as Red Hat Service Mesh.



Note

When your application interacts with critical external systems, such as payment providers, you must use mocks to avoid calling those systems.

The main advantage of this strategy is that you can do real-time user and performance testing without impact on the production version. Similar to other strategies, which duplicate the versions running at the same time, it increases the operational costs.



References

For more information about the `Deployment` and `DeploymentConfig` objects, refer to the *Understanding Deployment and DeploymentConfig objects* chapter in the *Red Hat OpenShift Container Platform Guide* at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html/applications/deployments#what-deployments-are

For more information about deployment strategies, refer to the chapter *Using deployment strategies* in the *Red Hat OpenShift Container Platform Guide* at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html/applications/deployments#deployment-strategies

Colorful deployments: An introduction to blue-green, canary, and rolling deployments

<https://opensource.com/article/17/5/colorful-deployments>

.

► Guided Exercise

Implementing Release Strategies

In this exercise you will deploy an application by using different deployment strategies.

The solution files are provided at the `greeting-service-v2` folder in the `solutions` branch of the `D0400-apps` repository. Notice that you might need to replace some strings in the `Jenkinsfile` file.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (```) instead of the backslash character (`\`) to introduce line breaks in these commands.

Outcomes

You should be able to deploy applications in OpenShift by using the Rolling and Recreate strategies.

Before You Begin

To perform this exercise, ensure you have:

- Your `D0400-apps` fork cloned in your workspace
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- A GitHub account
- The OpenShift CLI



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the deployment strategies of the `greeting-service` application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps/greeting-service` application folder and checkout the `main` branch of your `D0400-apps` fork, to ensure you start this exercise from a known clean state.


```
[user@host D0400]$ cd D0400-apps/greeting-service
[user@host greeting-service]$ git checkout main
...output omitted...
Your branch is up to date with 'origin/main'.
[user@host greeting-service]$ git pull origin main
...output omitted...
Already up to date.
```

- 1.2. Create a new branch to save any changes you make during this exercise. Push the branch to your GitHub fork.

```
[user@host greeting-service]$ git checkout -b deployment-strategies
Switched to a new branch 'deployment-strategies'
[user@host greeting-service]$ git push origin deployment-strategies
...output omitted...
* [new branch]      deployment-strategies -> deployment-strategies
```

- ▶ 2. Create a deployment by using the **Recreate** deployment strategy.

- 2.1. Log in to OpenShift with the credentials provided in the Lab Environment tab of the Red Hat online learning website. You can also copy the login command from the OpenShift web console.

```
[user@host greeting-service]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
[user@host greeting-service]$ oc new-project RHT_OCP4_DEV_USER-deploy-strategies
Now using project "youruser-deploy-strategies"
on server "https://api.cluster.example.com:6443".
...output omitted...
```

- 2.2. Use `oc new-app` to create a new Deployment resource in OpenShift for this application.

```
[user@host greeting-service]$ oc new-app --name greeting-service \
https://github.com/YOUR_GITHUB_USER/D0400-apps#deployment-strategies \
--context-dir=greeting-service \
--strategy=docker
...output omitted...
--> Success
Build scheduled, use 'oc logs -f bc/greeting-service' to track its progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/greeting-service'
Run 'oc status' to view your app.
```

The created application retrieves the code from the `deployment-strategies` branch. The `docker` build strategy builds the application image by using the `Dockerfile` found in the `greeting-service` folder in the `D0400-apps` repository. Note that, although build strategies control how a container image is built

and pushed, deployment strategies control how the pods of an application are rolled out in OpenShift.

- 2.3. Wait for the application to deploy. To check the status, run `oc get builds -w`. The `greeting-service-1` build is finished when the `STATUS` column is `Complete`.

```
[user@host greeting-service]$ oc get builds -w
NAME                TYPE    FROM          STATUS    ...output omitted...
greeting-service-1  Docker  Git@32ac625   Complete  ...output omitted...
```

Press **CTRL + c** to stop the command.

- 2.4. Expose the application by creating a route via the `expose` command.

```
[user@host greeting-service]$ oc expose svc/greeting-service
route.route.openshift.io/greeting-service exposed
```

- 2.5. Get the route of the application.

```
[user@host greeting-service]$ oc get route greeting-service \
-o jsonpath="{.spec.host}"{'\n'}"
greeting-service-your-user-deploy-strategies.apps.example.com
```

Copy this URL. You will use it in subsequent steps.

- 2.6. In a web browser, use the URL that you just retrieved to navigate to the application. Verify that the application response body is `Hello guest!`.
- 2.7. Change the newly created deployment to use the **Recreate** deployment strategy.

```
[user@host greeting-service]$ oc edit deployment greeting-service
```

Modify the `spec.strategy` object to only include the `type` property set to `Recreate`. Remove the rest of properties within the `spec.strategy` object.

```
...output omitted...
spec:
  ...output omitted...
  strategy:
    type: Recreate
  template:
    ...output omitted...
```

Save the changes.

- ▶ 3. Configure the deployment stage by using a Jenkinsfile.

- 3.1. By using your editor of choice, open the `Jenkinsfile` file and modify its contents to deploy the application to OpenShift. Replace the file contents with the following pipeline:

```
pipeline{
  agent any

  stages{
```

```

    stage('Deploy') {
        steps {
            sh '''
                oc project RHT_OCP4_DEV_USER-deploy-strategies
                oc start-build greeting-service --follow --wait
            '''
        }
    }
}

```

Replace `RHT_OCP4_DEV_USER` with your OpenShift user name and save the file.

3.2. Commit and push the changes.

```

[user@host greeting-service]$ git commit -a -m "Replace Jenkinsfile"
[deployment-strategies 60947f2] Replace Jenkinsfile
1 file changed, 14 insertions(+), 26 deletions(-)
rewrite greeting-service/Jenkinsfile (87%)
[user@host greeting-service]$ git push origin deployment-strategies
...output omitted...
* [new branch]      deployment-strategies -> deployment-strategies

```

3.3. Allow Jenkins to start builds in the `RHT_OCP4_DEV_USER-deploy-strategies` project, by using the `oc policy add-role-to-user` command. Because the currently selected project is `RHT_OCP4_DEV_USER-deploy-strategies`, the permission is only added for this project.

```

[user@host greeting-service]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"

```

► 4. Open Jenkins and create a new pipeline for the `greeting-service` application.

- 4.1. Open a new browser tab, navigate to the URL of your Jenkins instance and login with your OpenShift credentials.
- 4.2. Click **New Item** in the left navigation pane. Enter `greeting-service-deployment` as the item name, select **Pipeline** and click **OK**.
- 4.3. In the **Pipeline** section, specify the following configuration:

Field	Value
Definition	Pipeline script from SCM
SCM	Git
Repository URL	<code>https://github.com/YOUR_GITHUB_USER/DO400-apps/</code>
Branch Specifier	<code>deployment-strategies</code>
Script Path	<code>greeting-service/Jenkinsfile</code>

- 4.4. Click **Save**. In the pipeline details page, click **Build now**. Jenkins triggers a new build for the new pipeline.
- 4.5. After the pipeline finishes, check the build logs, which should show a successful build.

```
...output omitted...
Push successful
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

- 4.6. Go to the application URL and check that the response is still `Hello guest!`.

► **5.** Introduce a change in the application and deploy the application. Validate that the **Recreate** deployment strategy causes downtime.

- 5.1. Open the `greet.js` file and modify the greeting message as follows:

```
module.exports = function greet(name) {
  return `Hello ${name || "guest"} from version 2!`;
}
```

Save the file.

- 5.2. Commit and push the changes.

```
[user@host greeting-service]$ git commit -a -m "Modify greeting message"
[deployment-strategies 78c6d7f] Modify greeting message
1 file changed, 1 insertion(+), 1 deletion(-)
[user@host greeting-service]$ git push origin deployment-strategies
...output omitted...
To https://github.com/your_github_user/D0400-apps.git
4baa4e0..78c6d7f deployment-strategies -> deployment-strategies
```

- 5.3. Open a new terminal and navigate to the `D0400-apps` folder. Run the `tools/check.py` script, to continuously check the application URL. Replace `APPLICATION_URL` with the application URL that you retrieved before.

```
[user@host ~]$ cd D0400/D0400-apps
[user@host D0400-apps]$ python tools/check.py http://APPLICATION_URL
Checking http://greeting-service-your-user-deploy-
strategies.apps.cluster.example.com/

2020-11-26 17:03:16 INFO      Hello guest!
2020-11-26 17:03:18 INFO      Hello guest!
...output omitted...
```

Leave this terminal open.

- 5.4. Return to the browser window and navigate to the pipeline details page. Click **Build Now** to schedule a pipeline run. This will cause the deployment of your new greeting message.
- 5.5. Open the terminal where the `check.py` script is running. Verify that, after Jenkins triggers the new deployment, some of your requests are rejected with the HTTP Error 503: Service Unavailable error until the new version is deployed.

```
...output omitted...
2020-11-26 17:22:15 INFO      Hello guest!
2020-11-26 17:22:16 ERROR    HTTP Error 503: Service Unavailable
...output omitted...
2020-11-26 17:22:27 ERROR    HTTP Error 503: Service Unavailable
2020-11-26 17:22:28 INFO      Hello guest from version 2!
...output omitted...
```

Also, note how the old and the new version do not run along each other at the same time. This is because the Recreate strategy does not deploy the new version until the old version has stopped completely, therefore causing downtime.

Do not stop the `check.py` script. You will use it in the next step.

- ▶ 6. Change the deployment strategy to **Rolling** and deploy a new version. Verify that the deployment does not cause downtime.

- 6.1. Edit the Deployment resource to use the **Rolling** deployment strategy.

```
[user@host greeting-service]$ oc edit deployment greeting-service
```

Modify the `spec.strategy` object to only include the `type` property set to `Rolling`.

```
...output omitted...
spec:
  ...output omitted...
  strategy:
    type: RollingUpdate
  template:
  ...output omitted...
```

- 6.2. Open the `greeting-service/greet.js` file and modify the greeting message as follows:

```
module.exports = function greet(name) {
  return `Hello ${name || "guest"} from V3!`;
}
```

Save the file.

- 6.3. Commit and push the changes.

```
[user@host greeting-service]$ git commit -a -m "Modify greeting message for v3"
[deployment-strategies 4dcdbb] Modify greeting message for v3
1 file changed, 1 insertion(+), 1 deletion(-)
[user@host greeting-service]$ git push origin deployment-strategies
...output omitted...
To https://github.com/your_github_user/D0400-apps.git
3789a2d..4dcdbb deployment-strategies -> deployment-strategies
```

- 6.4. Return to the browser window and navigate to the pipeline details page. Click **Build Now** to schedule a pipeline run. This will cause a new deployment.
- 6.5. Open the terminal where the `check.py` script is running. Verify that, after Jenkins rolls out the deployment, there is no downtime before version 3 becomes available.

```
...output omitted...
2020-11-26 17:40:28 INFO      Hello guest from version 2!
2020-11-26 17:40:30 INFO      Hello guest from V3!
...output omitted...
```

In this case, the Rolling deployment strategy does not scale down version 2 until version 3 becomes available. With this strategy, it is possible that both the old and new versions coexist until the deployment is completed.

- 6.6. Press **CTRL + c** to stop the `check.py` script.

► 7. Clean up.

- 7.1. Delete the `RHT_OCP4_DEV_USER-deploy-strategies` project and navigate back to your workspace folder.

```
[user@host greeting-service]$ oc delete project \
RHT_OCP4_DEV_USER-deploy-strategies
project.project.openshift.io "your-user-deploy-strategies" deleted
[user@host greeting-service]$ cd ../../
[user@host D0400]$
```

- 7.2. Return to the Jenkins tab in your browser, and click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.

This concludes the guided exercise.

Executing Automated Tests in Pipelines

Objectives

After completing this section, you should be able to configure pipelines to automatically test an application to lower risk for production outages.

Defining Continuous Testing

Automated testing and CI are closely related practices, which you usually must apply together. It makes little sense to configure a CI pipeline without tests because the main purpose of CI is to validate new changes. If you had a pipeline without tests then you would be integrating changes into the `main` branch without any verification and potential errors could easily leak into production. This makes the development process more expensive because bugs that reach production have a higher cost.

Likewise, if you have a suite of automated tests, but only run them in local development environments, then tests will stop being an essential part of CI and change management. Consequently, if the team stops maintaining the tests, at some point the tests will break, and the team will eventually abandon them making your initial testing efforts useless. In general, automated tests lose their value if you do not run them continuously.

Executing your automated tests in a CI pipeline lowers the risk of production failures and keeps your tests alive and valuable as an active part of the delivery process. In this way, the pipeline executes the tests frequently, usually after you push changes to the repository, and helps your team to achieve the DevOps goal of a high-quality, responsive delivery process with short feedback loops. This practice is called *Continuous testing*.

Failing Fast

DevOps encourages the *fail fast* mentality to shorten feedback loops. The order in which you execute your tests in a CI pipeline matters.

For example, assume that you execute your unit tests in the CI pipeline after a set of slow code checks, which takes five minutes. If you push a change that breaks the tests, then you will have to wait at least five minutes to realize that the tests fail, and you will also waste hardware resources. Because of this, you should place **fast tasks earlier in your pipeline**.

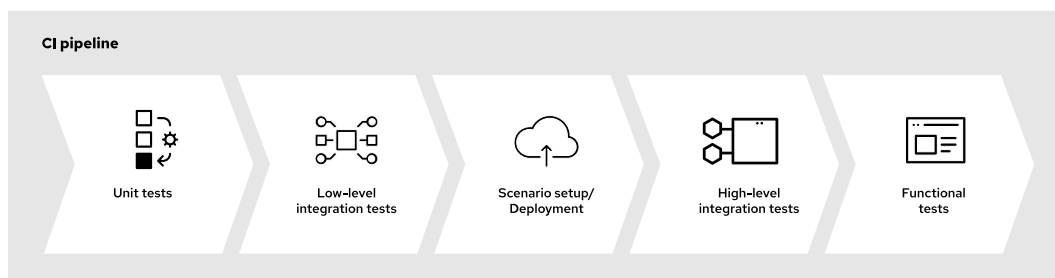


Figure 6.6: Tests order in a CI pipeline

Although the test order in a pipeline can differ for each project, generally you can sort your pipeline stages following this order:

1. **Unit tests.** Unit tests run very quickly, so they should be one of the first tasks to run in the pipeline. They are easy to set up, and you can run them in pipelines generally without issues. The stages that precede unit tests should only check out code and install dependencies necessary for the tests to run. Additionally, it is common to run quick static code analyses in parallel to unit tests.
2. **Integration tests.** Run integration tests after unit tests. Integration tests take more time to set up and run than unit tests. Depending on the level at which integration tests operate, their duration might vary. For example, testing the integration of two low-level classes can be as quick as a unit test. In contrast, testing the high-level integration of a component with an external system, such as a database, is slower and requires a more complex scenario setup.
3. **Functional tests.** Functional tests take more resources and longer time to run. More extensive test suites could take a long time, in some cases a half hour or more. Because of this, it is sometimes preferred to only run functional tests after integrating code into the `main` branch. Place functional tests towards the end of the pipeline, generally after having deployed the application that you want to test.

This order is the same order as you ascend the testing pyramid.

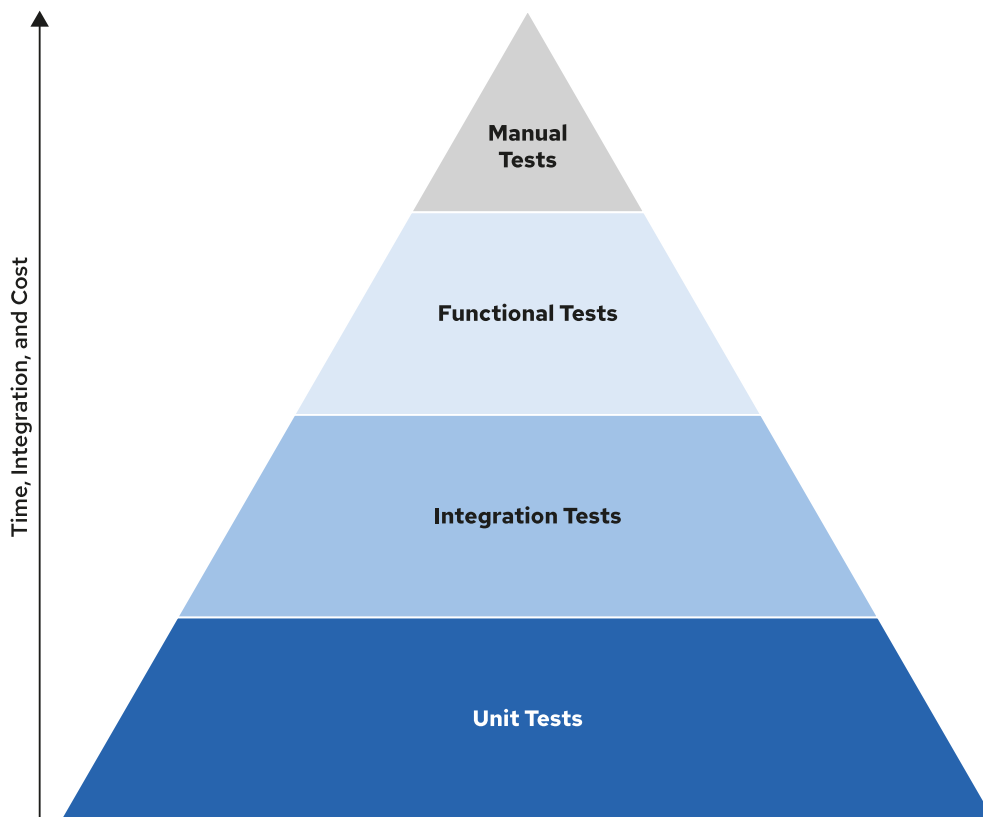


Figure 6.7: The testing pyramid

Tests on higher levels of the pyramid are normally slower and more difficult to set up and maintain, as seen in *Describing the Testing Pyramid*. High-level integration and functional tests generally provide very valuable feedback, but CI becomes more complex when you add these tests to the pipeline. Carefully choose your integration and functional tests. You should test what is *most*

valuable for your team. Blindly testing everything could overwhelm your team with testing and CI maintenance tasks.

Running Tests in CI Pipelines

To run your tests in a pipeline, the agents running the pipeline must have the required runtimes and tools available. For example, if you are testing a Node.js application, then you need an agent with Node.js installed.

To run unit tests, use an agent with the required runtime and install the libraries or packages required by your application. The installation of application dependencies is usually included as part of the pipeline, by using runtime-specific tools, such as Maven or NPM, as shown in the following example:

```
pipeline {
  agent any
  stages {
    stage('Install Dependencies') {
      steps {
        sh 'npm ci'
      }
    }
    stage('Unit Test') {
      steps {
        sh 'npm test'
      }
    }
  }
}
```

As you go up in the testing pyramid, you need more efforts to set up and introduce the tests in a CI pipeline. With integration tests, you must **execute multiple components** to verify their correct integration. The setup complexity depends on the level at which your integration tests run. Low-level integration tests might be as easy to set up as unit tests, although higher levels might be closer to functional testing setups. For example, if you are testing the integration of two microservices, then you must set up a test scenario in which both microservices are running.

Functional tests are the hardest to set up, because they require a near fully operational system to test the application from the user perspective. For example, with browser-based functional tests, the Jenkins agent needs a browser capable of running in a non-graphical environment, and possibly specific libraries installed at the operating system level. Most functional testing libraries provide container images preconfigured with all the required dependencies to run the tests.

Using Agents as Test Execution Environments

To run your tests in a pipeline, you must specify the agent that will run the whole pipeline or stage-specific agents.

Unit tests only need an agent with the required runtime. Select the agent in the `agent` section of the Jenkinsfile, by using the `node` and `label` directives.

```

pipeline {
  agent {
    node { label 'nodejs' }
  }
  stages {
    stage('Install Dependencies') {
      steps {
        sh 'npm ci'
      }
    }
    stage('Unit Test') {
      steps {
        sh 'npm test'
      }
    }
  }
}

```

This example shows how the whole pipeline uses agents labeled as `nodejs`.

The OpenShift Jenkins template that you use in this course instantiates a Jenkins application with the Kubernetes plug-in. This plug-in can dynamically provision Jenkins agents as OpenShift pods, based on the value provided by the `label` directive.

If you use the `agent` any directive, then Jenkins uses the main controller node to run the pipeline, which includes basic runtimes, such as Java and Python. By default, OpenShift also provides the Node.js and Maven agent images, which you can select by using the `label 'nodejs'` and `label 'maven'` directives respectively.



Note

If you need to create more complex agents to execute your test scenarios, then you must define your own custom agent images. Defining custom agent images is possible but outside the scope of this course.

For the stages that run more complex tests, you might want to use a specific agent.

```

pipeline {
  agent {
    node { label 'maven' }
  }
  stages {
    ...output omitted...

    stage("Functional Tests") {
      agent { node { label 'nodejs-cypress' } }
      steps {
        sh 'npm run test:functional'
      }
    }
  }
}

```

The preceding example shows how the stage for functional tests overrides the Maven agent, by using an agent suitable to run Cypress functional tests.



Note

Jenkins also supports alternative ways to use container images as execution environments within pipelines, such as the Docker Pipeline Plug-in. This plug in requires Docker installed in the main Jenkins controller, which is not the case in this course.



References

What you need to know about automation testing in CI/CD

<https://opensource.com/article/20/7/automation-testing-cicd>

Jenkins Docker Image GitHub Repository

<https://github.com/openshift/jenkins>

For more information, refer to the *Configuring Jenkins images* chapter in the *OpenShift Container Platform 4.6 Documentation* at

https://docs.openshift.com/container-platform/4.6/openshift_images/using_images/images-other-jenkins.html#images-other-jenkins-config-kubernetes_images-other-jenkins

► Guided Exercise

Executing Automated Tests in Pipelines

In this exercise you will configure a declarative pipeline that runs both unit and functional tests for a command-line interface (CLI) application.

The solution files are provided at the `exchange-cli` folder in the `solutions` branch of the `D0400-apps` repository. Notice that you might need to replace some strings in the `Jenkinsfile` file.

Outcomes

You should be able to run test suites by using Jenkins declarative pipelines.

Before You Begin

To perform this exercise, ensure you have:

- Node.js and NPM
- A web browser
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- The OpenShift CLI



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Create a new branch and run tests locally.
 - 1.1. Open a command-line terminal. Switch to the application directory in your cloned fork of the `D0400-apps` repository. Create a branch named `jenkins-testing`.

```
[user@host D0400]$ cd D0400-apps/exchange-cli
[user@host exchange-cli]$ git checkout -b jenkins-testing
Switched to a new branch 'jenkins-testing'
```

- 1.2. Install the Node.js dependencies.

```
[user@host exchange-cli]$ npm install
added 640 packages from 441 contributors and audited 640 packages in 5.958s

40 packages are looking for funding
  run npm fund for details

found 0 vulnerabilities
```

- 1.3. By using the included NPM scripts, run the unit tests locally. The tests should pass.

```
[user@host exchange-cli]$ npm run test:unit
...output omitted...
PASS src/convert.test.ts
  convert
    # should convert USD to USD (1 ms)
    # should convert from USD
    # should convert to USD
    # should convert between non-USD (1 ms)
    # should convert GBP to GBP

Test Suites: 1 passed, 1 total
Tests:      5 passed, 5 total
Snapshots:  0 total
Time:       0.518 s, estimated 1 s
Ran all test suites.
```

This test suite contains only unit tests, which are written in a "white box" fashion. These tests are written by using the Jest JavaScript testing framework.

- 1.4. By using the included NPM scripts, run the functional tests locally. The tests should pass.

```
[user@host exchange-cli]$ npm run test:functional
PASS ./functional.spec.ts
  functional tests
    miscellaneous flags
      # should print help (108 ms)
      # should check for missing flags (97 ms)
    conversions
      # should do a simple "conversion" (99 ms)
      # should convert different currencies (99 ms)

Test Suites: 1 passed, 1 total
Tests:      4 passed, 4 total
Snapshots:  0 total
Time:       0.907 s, estimated 3 s
Ran all test suites.
```

This test suite contains only functional tests, which are written in a "black box" fashion. These tests are also written by using the Jest testing framework, but they do *not* run alongside the unit tests.

See the chapter on writing tests for more information.

► 2. Create and commit a new declarative pipeline in a Jenkinsfile.

- 2.1. In the `D0400-apps/exchange-cli` directory, create a new file named `Jenkinsfile` with the following contents:

```
pipeline {
  agent {
    label 'nodejs'
  }

  stages {
    stage ('Install Dependencies') {
      steps {
        dir ('exchange-cli') {
          sh "npm install"
        }
      }
    }

    stage ('Build') {
      steps {
        dir ('exchange-cli') {
          sh "npm run build"
        }
      }
    }
  }
}
```

- 2.2. Commit the newly added Jenkinsfile and push the branch to your fork.

```
[user@host exchange-cli]$ git add Jenkinsfile
[user@host exchange-cli]$ git commit -m 'added basic pipeline in Jenkinsfile'
[jenkins-testing ca9862b] added basic pipeline in Jenkinsfile
1 file changed, 25 insertions(+), 0 deletions(-)
create mode 100644 exchange-cli/Jenkinsfile
[user@host exchange-cli]$ git push origin jenkins-testing
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 362 bytes | 362.00 KiB/s, done.
...output omitted...
```

► 3. Create a new pipeline job in Jenkins.

- 3.1. From the main Jenkins dashboard, click **New Item**.
- 3.2. Enter `exchange-cli` for the name, select **Pipeline** for the type, and click **OK**.
- 3.3. At the bottom of the form, select or enter the following:

Field	Value
Definition	Pipeline script from SCM
SCM	Git
Repository URL	https://github.com/ YOUR_GITHUB_USERNAME/DO400- apps.git.
Branch Specifier	*/jenkins-testing
Script Path	exchange-cli/Jenkinsfile

**Note**

The Repository URL is found on your fork's GitHub page by clicking **Clone**.
Be sure to select the HTTPS clone URL.

Click **Save** to save the changes.

3.4. From the menu on the left, click **Build Now**. The build should finish successfully:

```
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS
```

► **4.** Add a pipeline stage to run unit tests.

4.1. Edit the Jenkinsfile and add a **Unit Tests** stage after the **Build** stage.

```
...output omitted...
stages {
  stage ('Build') {
    ...output omitted...
  }

  stage ('Unit Tests') {
    steps {
      dir ('exchange-cli') {
        sh "npm run test:unit"
      }
    }
  }
}
```

4.2. Commit the changes and push to the branch. In Jenkins, run the pipeline again and open the output.

The unit tests should run and pass:

```

...output omitted...
[Pipeline] { (Unit Tests)
[Pipeline] dir
Running in /tmp/workspace/exchange-cli-jenkinsfile/exchange-cli
[Pipeline] {
[Pipeline] sh
+ npm run test:unit
...output omitted...
PASS src/convert.test.ts
  convert
    ### should convert USD to USD (1 ms)
    ### should convert from USD
    ### should convert to USD
    ### should convert between non-USD
    ### should convert GBP to GBP (1 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        3.105 s
Ran all test suites.
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS

```

**Note**

It is normal for the check mark symbols output by Jest to show up as question marks in the Jenkins console output.

► **5.** Add a pipeline stage to run functional tests.

- 5.1. Edit the Jenkinsfile and add a `Functional Tests` stage after the `Unit Tests` stage.

```

stages {
  ...output omitted...
  stage ('Unit Tests') {
    ...output omitted...
  }
  stage ('Functional Tests') {
    steps {
      dir ('exchange-cli') {
        sh "npm run test:functional"
      }
    }
  }
}

```

- 5.2. Commit the changes and push to the branch. In Jenkins, run the pipeline for a final time and check the console output. Both the unit and functional tests should run and pass:


```

...output omitted...
[Pipeline] { (Functional Tests)
[Pipeline] dir
Running in /tmp/workspace/exchange-cli-jenkinsfile/exchange-cli
[Pipeline] {
[Pipeline] sh
+ npm run test:functional
...output omitted...
PASS ./functional.spec.ts
  functional tests
    miscellaneous flags
      ### should print help (124 ms)
      ### should check for missing flags (116 ms)
    conversions
      ### should do a simple "conversion" (119 ms)
      ### should convert different currencies (136 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        3.952 s
Ran all test suites.
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS

```

- ▶ **6.** Clean up. Click **Back to Project**, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.

This concludes the guided exercise.

► Lab

Deploying Applications with Pipelines

In this lab, you will develop a CI/CD pipeline to automatically test, build, and deploy an application.

The solution files are provided at the `home-automation-service` folder in the `solutions` branch of the `DO400-apps` repository. Notice that you might need to replace some strings in the `Jenkinsfile` file.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (```) instead of the backslash character (`\`) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Introduce automated tests in the initial stages of your pipelines.
- Use CI/CD pipelines to build application container images and publish them to a container registry.
- Use CI/CD pipelines to deploy to multiple environments based on the Git branch.

Before You Begin

To perform this exercise, ensure you have the following:

- Your `DO400-apps` fork cloned in your workspace
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- A GitHub account
- The OpenShift CLI

You will be using a Quarkus application that controls a home lighting system, based on environment conditions. The source code for the application is available in the `home-automation-service` folder in the GitHub repository at <https://github.com/RedHatTraining/DO400-apps>. To verify that the pipeline deploys the application successfully, you will check that the `/home/lights` endpoint of the application returns the `Lights are off` response.

Also, make sure you start this lab from the `DO400-apps` application folder. Checkout the `main` branch to start this lab from a known clean state.

**Important**

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
```

Instructions

1. Create a new public GitHub repository named `do400-deploying-lab` with the contents of the `D0400-apps/home-automation-service` folder. Initialize the repository locally and push the code to the GitHub repository.
2. Create a `Jenkinsfile` file in the root folder of your local repository and add a `Test` stage to run the unit tests of the application. The pipeline must use the `maven` agent label and execute the tests by using the `./mvnw verify` command. Commit the `Jenkinsfile` file and push the changes to GitHub.
3. Create a personal access token in GitHub and configure a `Github` server in your Jenkins instance to allow Jenkins to manage GitHub hooks. The hooks allow GitHub to notify Jenkins when there is a change in the repository.

Create the GitHub token with the `repo`, `admin:repo_hook`, `read:user`, and `user:email` permissions. The name of the token will not affect the exercise, but you can name the token `do400-deploying-lab` for clarity. Make sure that Jenkins has a GitHub Server configured to register GitHub hooks by selecting the **Manage hooks** check box.

**Warning**

Make sure to copy the generated GitHub token. The token is only displayed once and you will need it in the following steps.

**Note**

You can use the personal access token that you created in the previous exercise of this chapter. If you decide to use an existing token, then make sure it includes the right permissions.

4. Configure a new pipeline for the `do400-deploying-lab` GitHub repository in your Jenkins instance. Use the Blue Ocean interface for this. After adding the repository as a pipeline, verify that the repository has a webhook registered. Finally, check that the pipeline first execution runs the `Test` stage successfully.
5. Add your Quay.io credentials in Jenkins to publish images from pipelines. Create a new credential in the Jenkins store and `Global credentials (unrestricted)` domain. Name the credential `QUAY_USER` and choose the `Username with password` kind.

**Note**

You do not need to follow this step if you have already configured the Quay.io credentials for Jenkins in a previous guided exercise. In this case, make sure that the credential is called `QUAY_USER` or adjust the name in the pipeline accordingly.

6. Add a **Build & Push Image** stage to the Jenkins pipeline. This stage builds a container image from the Quarkus application and deploys the image to Quay.io.

- Expose the `QUAY_USER` credentials, which you created in the previous step, in the `Jenkinsfile` file. To expose the `QUAY_USR` and `QUAY_PSW` variables in the pipeline, you can use the `environment` directive and the `credentials` helper as the following example shows:

```
environment { QUAY = credentials('QUAY_USER') }
```

- To build and push the image, you can use the `quarkus-container-image-jib` extension in the pipeline as follows:

```
sh './mvnw quarkus:add-extension -Dextensions="container-image-jib"'
```

- Call the `./mvnw package` command in your pipeline to build the image and publish it in Quay.io. Use the `quay.io/redhattraining/do400-java-alpine-openjdk11-jre:latest` image as the base image. You must tag the published image as `latest` and `build-${BUILD_NUMBER}`. Including the Jenkins build number in the image tag will allow you to rollout deployments in each pipeline build.

```
sh '''
    ./mvnw package -DskipTests \
        -Dquarkus.jib.base-jvm-image=quay.io/redhattraining/do400-java-alpine-
openjdk11-jre:latest \
        -Dquarkus.container-image.build=true \
        -Dquarkus.container-image.registry=quay.io \
        -Dquarkus.container-image.group=$QUAY_USR \
        -Dquarkus.container-image.name=do400-deploying-lab \
        -Dquarkus.container-image.username=$QUAY_USR \
        -Dquarkus.container-image.password="$QUAY_PSW" \
        -Dquarkus.container-image.tag=build-${BUILD_NUMBER} \
        -Dquarkus.container-image.additional-tags=latest \
        -Dquarkus.container-image.push=true
'''
```

After you have made all the necessary changes to the `Jenkinsfile` file, commit and push your changes. Wait for the pipeline to finish successfully and verify that the image has been pushed to your Quay.io account.

Finally, make your Quay.io image public. You will use this image to deploy the application in the next steps.

7. Create a new *TEST* environment as a Red Hat OpenShift project and deploy the application to this testing environment.
 - Create the environment as a new OpenShift project called *RHT_OCP4_DEV_USER-deploying-lab-test*. Replace *RHT_OCP4_DEV_USER* with the username provided in the **Lab Environment** tag of the course's online learning website.
 - Use the `kubefiles/application-template.yml` template to deploy the application using the image that you just pushed to Quay.io. Apply the template in the *RHT_OCP4_DEV_USER-deploying-lab-test* project as the namespace and specify your Quay.io user, as shown in this example:

```
[user@host deploying-lab]$ oc process \
-n RHT_OCP4_DEV_USER-deploying-lab-test \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_IMAGE_NAME=do400-deploying-lab \
-p APP_IMAGE_TAG=latest \
| oc apply -n RHT_OCP4_DEV_USER-deploying-lab-test -f -
```

After deploying the application, navigate to `http://YOUR-TEST-URL/home/lights`. and verify that the application response is `Lights are off`.

8. Add a **Deploy to Test** stage in the `Jenkinsfile` file to continuously deploy the image to the *TEST* environment.
 - Use the `oc set image deployment home-automation` command to change the image tag of the `home-automation` deployment, by using the Jenkins build number. This will roll out the image pushed in the **Build & Push Image** stage. You must update the image tag for the `home-automation` container, and run the command in the *RHT_OCP4_DEV_USER-deploying-lab-test* namespace. You can also use the `--record` flag for later rollout inspection.

```
sh """
  oc set image deployment home-automation \
  home-automation=quay.io/${QUAY_USR}/do400-deploying-lab:build-${BUILD_NUMBER}
  \
  -n RHT_OCP4_DEV_USER-deploying-lab-test --record
"""
```

- Only run this stage if the branch is not `main`. Create this stage in a Git branch called `deployments` to verify that Jenkins executes the pipeline.

After you make all the necessary changes to the `Jenkinsfile` file, commit and push the changes to trigger a deployment to *TEST*. Verify that Jenkins has rolled out a deployment to *TEST*.



Important

Jenkins requires explicit permission to run deployments in the *TEST* project of OpenShift. You must grant the `edit` role to the Jenkins account in the *TEST* project. To this end, use the `oc policy add-role-to-user` command.

9. Create a new *PROD* environment as an OpenShift project and deploy the application to this production environment.
 - Create the *PROD* environment as a new OpenShift project called *RHT_OCP4_DEV_USER-deploying-lab-prod*. Replace *RHT_OCP4_DEV_USER* with the username provided in the **Lab Environment** tag of the course's online learning website.
 - Use the `kubefiles/application-template.yml` template to deploy the application using the image that you just pushed to Quay.io. Apply the template in the *RHT_OCP4_DEV_USER-deploying-lab-prod* project as the namespace and your Quay.io user, as shown in this example:

```
[user@host deploying-lab]$ oc process \
-n RHT_OCP4_DEV_USER-deploying-lab-prod \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_IMAGE_NAME=do400-deploying-lab \
-p APP_IMAGE_TAG=latest \
| oc apply -n RHT_OCP4_DEV_USER-deploying-lab-prod -f -
```

**Note**

Usually you should use different image tags for different environments to avoid conflicts. This lab uses the same `1.0-SNAPSHOT` tag for both environments for the sake of simplicity.

After deploying the application, navigate to `http://YOUR-PROD-URL/home/lights`. and verify that the application response is **Lights are off**.

10. Add a **Deploy to PROD** stage in the `Jenkinsfile` file to continuously deploy the image to the *PROD* environment.
 - Use the `oc set image deployment home-automation` command to change the image tag of the `home-automation` production deployment, by using the Jenkins build number. This will roll out the image pushed in the **Build & Push Image** stage. You must update the image tag for the `home-automation` container, and run the command in the

RHT_OCP4_DEV_USER-deploying-lab-prod namespace. You can also use the `--record` flag for later rollout inspection.

```
sh """
  oc set image deployment home-automation \
    home-automation=quay.io/${QUAY_USR}/do400-deploying-lab:build-${BUILD_NUMBER}
  \
  -n RHT_OCP4_DEV_USER-deploying-lab-prod --record
"""
```

- Only run this stage if the branch is `main`.

After you make all the necessary changes to the `Jenkinsfile` file, commit and push the changes to trigger a deployment to *PROD*. Verify that Jenkins does not run the `Deploy to PROD` in the `deployments` branch



Important

Jenkins requires explicit permission to run deployments in the *PROD* project of OpenShift. You must grant the `edit` role to the Jenkins account in the *PROD* project. To this end, use the `oc policy add-role-to-user` command.

11. Merge the `deployments` branch into the `main` branch and verify that Jenkins runs the deployment to the *PROD* environment in the `main` branch.



Note

Normally, you should create a pull request to propose your branch for integration into the `main` branch. In this case, we use the `git merge` command for the sake of simplicity.

12. Clean up. Delete the *TEST* and *PROD* environments, and the Jenkins pipeline project.

This concludes the lab.

► Solution

Deploying Applications with Pipelines

In this lab, you will develop a CI/CD pipeline to automatically test, build, and deploy an application.

The solution files are provided at the `home-automation-service` folder in the `solutions` branch of the `D0400-apps` repository. Notice that you might need to replace some strings in the `Jenkinsfile` file.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (```) instead of the backslash character (`\`) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Introduce automated tests in the initial stages of your pipelines.
- Use CI/CD pipelines to build application container images and publish them to a container registry.
- Use CI/CD pipelines to deploy to multiple environments based on the Git branch.

Before You Begin

To perform this exercise, ensure you have the following:

- Your `D0400-apps` fork cloned in your workspace
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- A GitHub account
- The OpenShift CLI

You will be using a Quarkus application that controls a home lighting system, based on environment conditions. The source code for the application is available in the `home-automation-service` folder in the GitHub repository at <https://github.com/RedHatTraining/DO400-apps>. To verify that the pipeline deploys the application successfully, you will check that the `/home/lights` endpoint of the application returns the `Lights are off` response.

Also, make sure you start this lab from the `D0400-apps` application folder. Checkout the `main` branch to start this lab from a known clean state.

**Important**

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start the guided exercise from this folder.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
```

Instructions

1. Create a new public GitHub repository named `do400-deploying-lab` with the contents of the `D0400-apps/home-automation-service` folder. Initialize the repository locally and push the code to the GitHub repository.
 - 1.1. Copy the contents of the `D0400-apps/home-automation-service` folder to a new `deploying-lab` folder in your workspace.

```
[user@host D0400-apps]$ cp -Rv home-automation-service \
~/D0400/deploying-lab
...output omitted...
```

**Important**

Windows users must also run the preceding command but in PowerShell as follows:

```
PS C:\Users\user\D0400\D0400-apps> CopyItem -Path home-automation-service `
>> -Destination ~\D0400\deploying-lab -Recurse
```

- 1.2. Create an empty public repository in GitHub named `do400-deploying-lab` to host the source code of the home automation application. In GitHub, click + → **New repository** in the upper right of the window. Enter `do400-deploying-lab` for the repository name and do not change any other field. Click **Create repository**.
- 1.3. Initialize a local repository in the `~/D0400/deploying-lab` folder and add the remote origin.

```
[user@host D0400-apps]$ cd ~/D0400/deploying-lab
[user@host deploying-lab]$ git init .
Initialized empty Git repository in ...output omitted.../D0400/deploying-lab/.git/
[user@host deploying-lab]$ git remote add origin \
https://github.com/YOUR_GITHUB_USER/do400-deploying-lab.git
```

Replace `YOUR_GITHUB_USER` with your GitHub username.

- 1.4. Commit the code, set `main` as the default branch, and push the code to the `origin` remote repository.

```
[user@host deploying-lab]$ git add -A
```

**Important**

Windows users must also execute the following command after the `git add` command:

```
PS C:\Users\user\DO400\deploying-lab> git update-index --chmod=+x mvnw
```

```
[user@host deploying-lab]$ git commit -m "Initial commit"
[master (root-commit) ecf7586] Initial commit
 29 files changed, 1335 insertions(+)
...output omitted...
[user@host deploying-lab]$ git branch -M main
[user@host deploying-lab]$ git push origin main
...output omitted...
To github.com:your_github_user/do400-deploying-lab.git
 * [new branch]      main -> main
```

If Git asks for credentials, then provide your username and personal access token.

2. Create a `Jenkinsfile` file in the root folder of your local repository and add a `Test` stage to run the unit tests of the application. The pipeline must use the `maven` agent label and execute the tests by using the `./mvnw verify` command. Commit the `Jenkinsfile` file and push the changes to GitHub.

- 2.1. In the root of the `deploying-lab` folder, create a file named `Jenkinsfile` with the following contents:

```
pipeline {
    agent {
        node { label "maven" }
    }

    stages {
        stage("Test") {
            steps {
                sh "./mvnw verify"
            }
        }
    }
}
```

- 2.2. Stage and commit the `Jenkinsfile` file to the local repository.

```
[user@host deploying-lab]$ git add Jenkinsfile
[user@host deploying-lab]$ git commit -m "Added Jenkinsfile"
[main eb6ad6c] Added Jenkinsfile
 1 file changed, 13 insertions(+)
 create mode 100644 Jenkinsfile
```

- 2.3. Push the local commits to the remote GitHub repository.

```
[user@host deploying-lab]$ git push --set-upstream origin main
...output omitted...
To https://github.com/your_github_user/do400-deploying-lab.git
 97c6b69..0ff54e8      HEAD -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

3. Create a personal access token in GitHub and configure a `Github` server in your Jenkins instance to allow Jenkins to manage GitHub hooks. The hooks allow GitHub to notify Jenkins when there is a change in the repository.

Create the GitHub token with the `repo`, `admin:repo_hook`, `read:user`, and `user:email` permissions. The name of the token will not affect the exercise, but you can name the token `do400-deploying-lab` for clarity. Make sure that Jenkins has a GitHub Server configured to register GitHub hooks by selecting the **Manage hooks** check box.



Warning

Make sure to copy the generated GitHub token. The token is only displayed once and you will need it in the following steps.



Note

You can use the personal access token that you created in the previous exercise of this chapter. If you decide to use an existing token, then make sure it includes the right permissions.

- 3.1. In a new browser tab, navigate to <https://github.com/settings/tokens>. You might need to authenticate with your GitHub username and password.
- 3.2. Click **Generate new token**. Name the token `do400-deploying-lab`. Select the `repo`, `admin:repo_hook`, `read:user` and `user:email` check boxes. Click **Generate token**.
- 3.3. In the **Personal access tokens** page, copy the newly created personal access token.
- 3.4. Return to the command-line terminal. Log in to OpenShift with the OpenShift CLI and get the URL of your Jenkins instance. Replace `RHT_OCP4_DEV_USER`, `RHT_OCP4_DEV_PASSWORD`, and `RHT_OCP4_MASTER_API` with the values provided in the **Lab Environment** tab of the course's online learning website.

```
[user@host deploying-lab]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
[user@host deploying-lab]$ oc get route \
-n RHT_OCP4_DEV_USER-jenkins jenkins \
-o jsonpath="{ 'http://'}{.spec.host}{ '\n'}"
http://jenkins-your-user-jenkins.apps.cluster.example.com
```

- 3.5. Open the Jenkins URL in a new browser tab. Log in into Jenkins with your OpenShift credentials, if necessary. Click **Manage Jenkins** in the left pane, and then click **Configure System** in the right pane.

- 3.6. Scroll down to the **GitHub** area to configure a GitHub server. Click **Add GitHub Server**.

**Note**

Do not add an additional GitHub server if you already created a server in a previous exercise.

- 3.7. Type **GitHub** in the **Name** field. In the **Credentials** group click **Add** and then select **Jenkins** to open a window, which allows you to add credentials.
- 3.8. In the **Jenkins Credentials Provider: Jenkins** window, select **Secret** text from the **Kind** list. In the **Secret** field, paste the personal token that you previously generated in GitHub. Type **GITHUB_TOKEN** in the **ID** field. Click **Add** to store the secret in Jenkins.

**Note**

Do not add an additional GitHub secret if you already created a secret in a previous exercise.

- 3.9. From the **Credentials** list select **GITHUB_TOKEN** and click **Test connection** to verify that the personal token has access to GitHub.
- 3.10. Select the **Manage hooks** check box.
- 3.11. Click **Save** to save the changes made to the Jenkins configuration.
4. Configure a new pipeline for the **do400-deploying-lab** GitHub repository in your Jenkins instance. Use the Blue Ocean interface for this. After adding the repository as a pipeline, verify that the repository has a webhook registered. Finally, check that the pipeline first execution runs the **Test** stage successfully.
- 4.1. In a browser tab, navigate to the Jenkins main page and click **Open Blue Ocean** in the left navigation pane.
- 4.2. In the Blue Ocean main page, click **New Pipeline** in the upper right.

**Important**

If there are no previously existing pipelines, Blue Ocean will show you the **Welcome to Jenkins** window. In that case, click **Create a new pipeline**.

- 4.3. In the **Where do you store your code?** step, select **GitHub**. If Jenkins asks for a personal access token, provide the access token that you created in the previous step.
- 4.4. In the **Which organization does the repository belong to?** step, select your GitHub user name.
- 4.5. In the **Choose a repository** step, select the **do400-deploying-lab** repository from the list and click **Create Pipeline**. After creating the pipeline, Jenkins triggers the first pipeline run.
- 4.6. Verify that the pipeline execution succeeds on the **main** branch. The Blue Ocean interface shows the success status as a green check mark.
- 4.7. Verify that Jenkins registered a webhook in the **do400-deploying-lab** repository. In a new browser tag, navigate to https://github.com/YOUR_GITHUB_USER/do400-deploying-lab/settings/hooks and verify that one webhook is displayed

on the page. A green check mark indicates that the last delivery of the hook has been successful.

5. Add your Quay.io credentials in Jenkins to publish images from pipelines. Create a new credential in the **Jenkins store** and **Global credentials (unrestricted)** domain. Name the credential **QUAY_USER** and choose the **Username with password** kind.



Note

You do not need to follow this step if you have already configured the Quay.io credentials for Jenkins in a previous guided exercise. In this case, make sure that the credential is called **QUAY_USER** or adjust the name in the pipeline accordingly.

- 5.1. Click **Administration** in the upper right of the window, and then click **Manage Credentials** in the right pane.
- 5.2. In the **Stores scoped to Jenkins** section, click **Jenkins**.
- 5.3. Click **Global credentials (unrestricted)** in the right pane, and then click **Add Credentials** in the left pane.
- 5.4. From the **Kind** list, select **Username with password**. In the **Username** field, type your Quay username. In the **Password** field, type your Quay password. In the **ID** field, type **QUAY_USER**, and then click **OK**.
6. Add a **Build & Push Image** stage to the Jenkins pipeline. This stage builds a container image from the Quarkus application and deploys the image to Quay.io.
 - Expose the **QUAY_USER** credentials, which you created in the previous step, in the **Jenkinsfile** file. To expose the **QUAY_USR** and **QUAY_PSW** variables in the pipeline, you can use the **environment** directive and the **credentials** helper as the following example shows:

```
environment { QUAY = credentials('QUAY_USER') }
```

- To build and push the image, you can use the **quarkus-container-image-jib** extension in the pipeline as follows:

```
sh './mvnw quarkus:add-extension -Dextensions="container-image-jib"'
```

- Call the **./mvnw package** command in your pipeline to build the image and publish it in Quay.io. Use the **quay.io/redhattraining/do400-java-alpine-openjdk11-jre:latest** image as the base image. You must tag the published image as **latest** and **build-\${BUILD_NUMBER}**. Including the Jenkins build number in the image tag will allow you to rollout deployments in each pipeline build.

```
sh '''
./mvnw package -DskipTests \
-Dquarkus.jib.base-jvm-image=quay.io/redhattraining/do400-java-alpine-
openjdk11-jre:latest \
-Dquarkus.container-image.build=true \
-Dquarkus.container-image.registry=quay.io \
-Dquarkus.container-image.group=$QUAY_USR \
-Dquarkus.container-image.name=do400-deploying-lab \
-Dquarkus.container-image.username=$QUAY_USR \
```

```

-Dquarkus.container-image.password="$QUAY_PSW" \
-Dquarkus.container-image.tag=build-${BUILD_NUMBER} \
-Dquarkus.container-image.additional-tags=latest \
-Dquarkus.container-image.push=true
...

```

After you have made all the necessary changes to the `Jenkinsfile` file, commit and push your changes. Wait for the pipeline to finish successfully and verify that the image has been pushed to your Quay.io account.

Finally, make your Quay.io image public. You will use this image to deploy the application in the next steps.

- 6.1. In your local workstation, open the `Jenkinsfile` file. Add the Quay.io credentials to the pipeline as an environment variable named `QUAY`. Add the `Build & Push Image` stage to the `Jenkinsfile` file after the `Test` stage, as follows:

```

pipeline {
    agent {
        node { label: "maven" }
    }

    environment { QUAY = credentials('QUAY_USER') }

    stages {
        ...output omitted...

        stage("Build & Push Image") {
            steps {
                sh '''
                    ./mvnw quarkus:add-extension \
                    -Dextensions="container-image-jib"
                '''
                sh '''
                    ./mvnw package -DskipTests \
                    -Dquarkus.jib.base-jvm-image=quay.io/redhattraining/do400-
java-alpine-openjdk11-jre:latest \
                    -Dquarkus.container-image.build=true \
                    -Dquarkus.container-image.registry=quay.io \
                    -Dquarkus.container-image.group=$QUAY_USR \
                    -Dquarkus.container-image.name=do400-deploying-lab \
                    -Dquarkus.container-image.username=$QUAY_USR \
                    -Dquarkus.container-image.password="$QUAY_PSW" \
                    -Dquarkus.container-image.tag=build-${BUILD_NUMBER} \
                    -Dquarkus.container-image.additional-tags=latest \
                    -Dquarkus.container-image.push=true
                '''
            }
        }
    }
}

```

Save the file.

- 6.2. Stage and commit the changes in the `Jenkinsfile` file to the local repository.

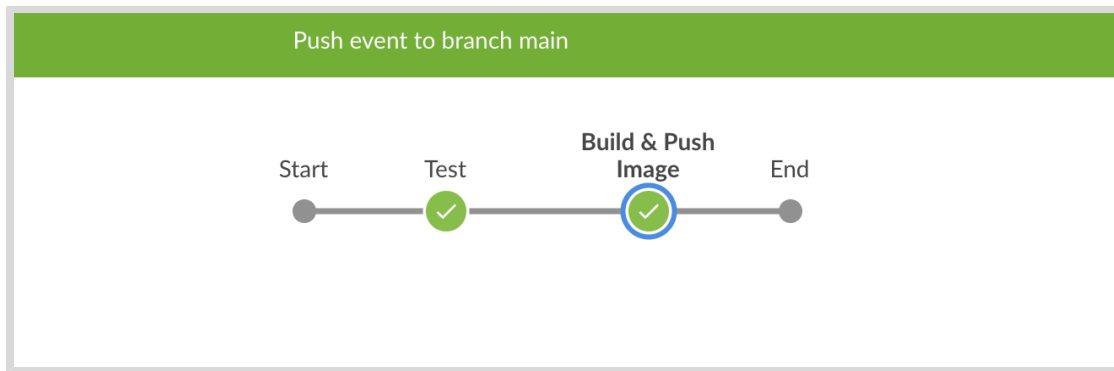
```
[user@host deploying-lab]$ git add Jenkinsfile
[user@host deploying-lab]$ git commit \
-m "Added pipeline stage to build and push image"
[main eb6ad6c] Added pipeline stage to build and push image
1 file changed, 21 insertions(+)
```

6.3. Push the new commit to the remote GitHub repository.

```
[user@host deploying-lab]$ git push
...output omitted...
To https://github.com/your_github_user/do400-deploying-lab.git
8f71c99..ed0dcf7  main -> main
```

6.4. Return to the Blue Ocean `do400-deploying-lab` browser tab. Navigate to the `do400-deploying-lab` project page and wait for the new pipeline build to start its execution. Click the last build to go to the build details page.

6.5. Wait for the pipeline to finish and verify that the execution is successful.



6.6. Open a new browser tab, navigate to <https://quay.io> and look for the newly created `do400-deploying-lab` repository. Click `do400-deploying-lab` to navigate to the details page of the `do400-deploying-lab` image repository. Alternatively, you can directly navigate to https://quay.io/repository/YOUR_QUAY_USER/do400-deploying-lab.

Click **Settings** in the left navigation pane. Scroll down the **Repository Visibility** section, click **Make Public** and click **OK** to confirm.

7. Create a new *TEST* environment as a Red Hat OpenShift project and deploy the application to this testing environment.
 - Create the environment as a new OpenShift project called `RHT_OCP4_DEV_USER-deploying-lab-test`. Replace `RHT_OCP4_DEV_USER` with the username provided in the **Lab Environment** tag of the course's online learning website.
 - Use the `kubefiles/application-template.yml` template to deploy the application using the image that you just pushed to Quay.io. Apply the template in the `RHT_OCP4_DEV_USER-deploying-lab-test` project as the namespace and specify your Quay.io user, as shown in this example:

```
[user@host deploying-lab]$ oc process \
-n RHT_OCP4_DEV_USER-deploying-lab-test \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_IMAGE_NAME=do400-deploying-lab \
-p APP_IMAGE_TAG=latest \
| oc apply -n RHT_OCP4_DEV_USER-deploying-lab-test -f -
```

After deploying the application, navigate to `http://YOUR-TEST-URL/home/lights`. and verify that the application response is `Lights are off`.

- 7.1. Return to the command-line terminal. Create a new Red Hat OpenShift project to host the *TEST* environment and switch to the new project.

```
[user@host deploying-lab]$ oc new-project \
RHT_OCP4_DEV_USER-deploying-lab-test
Now using project "youruser-deploying-lab-test" on server "https://
api.cluster.domain.example.com:6443".
...output omitted...
```

- 7.2. Create the deployment. Specify the project name for the *TEST* environment and your Quay.io user.

```
[user@host deploying-lab]$ oc process \
-n RHT_OCP4_DEV_USER-deploying-lab-test \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_IMAGE_NAME=do400-deploying-lab \
-p APP_IMAGE_TAG=latest \
| oc apply -n RHT_OCP4_DEV_USER-deploying-lab-test -f -
deployment.apps/home-automation created
service/home-automation created
route.route.openshift.io/home-automation created
```

- 7.3. Wait for the application to deploy. In order to check the status, run `oc get deployment -w` and wait until the `AVAILABLE` column is 1.

```
[user@host deploying-lab]$ oc get deployment -w
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
home-automation 1/1      1             1            44s
```

- 7.4. Get the route of the application.

```
[user@host deploying-lab]$ oc get route home-automation \
-o jsonpath='{\"http://\"}{.spec.host}{\"\\n\"}'
http://your-test-route.apps.cluster.example.com
```

- 7.5. Open a new browser tab and navigate to the `/home/lights` path of the application, as `YOUR_TEST_ROUTE/home/lights`. Check that the response is `Lights are off`.
8. Add a `Deploy` to `Test` stage in the `Jenkinsfile` file to continuously deploy the image to the *TEST* environment.

- Use the `oc set image deployment home-automation` command to change the image tag of the `home-automation` deployment, by using the Jenkins build number. This will roll out the image pushed in the `Build & Push Image` stage. You must update the image tag for the `home-automation` container, and run the command in the `RHT_OCP4_DEV_USER-deploying-lab-test` namespace. You can also use the `--record` flag for later rollout inspection.

```
sh """
  oc set image deployment home-automation \
    home-automation=quay.io/${QUAY_USR}/do400-deploying-lab:build-${BUILD_NUMBER}
  \
  -n RHT_OCP4_DEV_USER-deploying-lab-test --record
"""
```

- Only run this stage if the branch is not `main`. Create this stage in a Git branch called `deployments` to verify that Jenkins executes the pipeline.

After you make all the necessary changes to the `Jenkinsfile` file, commit and push the changes to trigger a deployment to `TEST`. Verify that Jenkins has rolled out a deployment to `TEST`.



Important

Jenkins requires explicit permission to run deployments in the `TEST` project of OpenShift. You must grant the `edit` role to the Jenkins account in the `TEST` project. To this end, use the `oc policy add-role-to-user` command.

- 8.1. Add the `edit` role to the Jenkins account in the `RHT_OCP4_DEV_USER-deploying-lab-test` project. Replace `RHT_OCP4_DEV_USER` with the `RHT_OCP4_DEV_USER` provided in the **Lab Environment** tag of the course's online learning website.

```
[user@host deploying-envs]$ oc policy add-role-to-user \
edit system:serviceaccount:`RHT_OCP4_DEV_USER`-jenkins:jenkins \
-n RHT_OCP4_DEV_USER-deploying-lab-test
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"
```

- 8.2. Create a branch named `deployments`.

```
[user@host deploying-lab]$ git checkout -b deployments
Switched to a new branch 'deployments'
```

- 8.3. Add a `Deploy to TEST` stage to your `Jenkinsfile` file, after the `Build & Push Image` stage. The stage only runs if the branch is not `main`. The `oc set image deployment` command changes the `home-automation` deployment, by updating the image tag of the `home-automation` container. Note that you must run the command in the `RHT_OCP4_DEV_USER-deploying-lab-test` project, which is the testing namespace.

```
stage('Deploy to TEST') {
  when { not { branch "main" } }
```

```

    steps {
      sh """
        oc set image deployment home-automation \
        home-automation=quay.io/${QUAY_USR}/do400-deploying-lab:build-
        ${BUILD_NUMBER} \
        -n RHT_OCP4_DEV_USER-deploying-lab-test --record
      """
    }
  }
}

```

8.4. Stage and commit the changes.

```

[user@host deploying-lab]$ git add Jenkinsfile
[user@host deploying-lab]$ git commit -m "Added test deployment"
[deployments f3780c0] Added test deployment
1 file changed, 12 insertions(+)

```

8.5. Push the local commits to the remote GitHub repository.

```

[user@host deploying-lab]$ git push --set-upstream origin deployments
...output omitted...
To https://github.com/your_github_user/do400-deploying-lab.git
 * [new branch]      deployments -> deployments
Branch 'deployments' set up to track remote branch 'deployments' from 'origin'.

```

8.6. Return to the Blue Ocean browser tab and verify that the pipeline is running in the `deployments` branch. Wait until the pipeline finishes and check that the output of the `Deploy to TEST` stage is equal to the following:

```

...output omitted...
deployment.apps/home-automation image updated

```

You can also run the `oc rollout history deployment home-automation` command to check that a second revision exists.

```

[user@host deploying-lab]$ oc rollout history deployment home-automation
deployment.apps/home-automation
REVISION  CHANGE-CAUSE
1          <none>
2          oc set image deployment home-automation ...output omitted...

```

9. Create a new *PROD* environment as an OpenShift project and deploy the application to this production environment.

- Create the *PROD* environment as a new OpenShift project called `RHT_OCP4_DEV_USER-deploying-lab-prod`. Replace `RHT_OCP4_DEV_USER` with the username provided in the **Lab Environment** tag of the course's online learning website.
- Use the `kubefiles/application-template.yml` template to deploy the application using the image that you just pushed to Quay.io. Apply the template in the `RHT_OCP4_DEV_USER-deploying-lab-prod` project as the namespace and your Quay.io user, as shown in this example:

```
[user@host deploying-lab]$ oc process \
-n RHT_OCP4_DEV_USER-deploying-lab-prod \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_IMAGE_NAME=do400-deploying-lab \
-p APP_IMAGE_TAG=latest \
| oc apply -n RHT_OCP4_DEV_USER-deploying-lab-prod -f -
```

Note

Usually you should use different image tags for different environments to avoid conflicts. This lab uses the same 1.0-SNAPSHOT tag for both environments for the sake of simplicity.

After deploying the application, navigate to <http://YOUR-PROD-URL/home/lights>. and verify that the application response is `Lights are off`.

- 9.1. Return to the command line terminal. Create a new OpenShift project to host the *PROD* environment and switch to the new project.

```
[user@host deploying-lab]$ oc new-project \
RHT_OCP4_DEV_USER-deploying-lab-prod
Now using project "youruser-deploying-lab-prod" on server "https://
api.cluster.domain.example.com:6443".
...output omitted...
```

- 9.2. Create the *PROD* deployment.

```
[user@host deploying-lab]$ oc process \
-n RHT_OCP4_DEV_USER-deploying-lab-prod \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p APP_IMAGE_NAME=do400-deploying-lab \
-p APP_IMAGE_TAG=latest \
| oc apply -n RHT_OCP4_DEV_USER-deploying-lab-prod -f -
deployment.apps/home-automation created
service/home-automation created
route.route.openshift.io/home-automation created
```

- 9.3. Wait for the application to deploy. In order to check the status, run `oc get deployment -w` and wait until the `AVAILABLE` column is 1.

```
[user@host deploying-lab]$ oc get deployment -w
NAME          READY  UP-TO-DATE  AVAILABLE  AGE
home-automation 1/1    1           1          20s
```

- 9.4. Get the route of the application.

```
[user@host deploying-lab]$ oc get route home-automation \
-o jsonpath='{\"http://\"}{.spec.host}{\"\\n\"}'
http://your-prod-route.apps.cluster.example.com
```

- 9.5. Open a browser tab and navigate to *YOUR_PROD_ROUTE*/home/lights. Check that the response is `Lights are off`.
10. Add a `Deploy` to `PROD` stage in the `Jenkinsfile` file to continuously deploy the image to the `PROD` environment.
 - Use the `oc set image deployment home-automation` command to change the image tag of the `home-automation` production deployment, by using the Jenkins build number. This will roll out the image pushed in the `Build & Push Image` stage. You must update the image tag for the `home-automation` container, and run the command in the `RHT_OCP4_DEV_USER-deploying-lab-prod` namespace. You can also use the `--record` flag for later rollout inspection.

```
sh """
  oc set image deployment home-automation \
    home-automation=quay.io/${QUAY_USR}/do400-deploying-lab:build-${BUILD_NUMBER}
  \
  -n RHT_OCP4_DEV_USER-deploying-lab-prod --record
"""
```

- Only run this stage if the branch is `main`.

After you make all the necessary changes to the `Jenkinsfile` file, commit and push the changes to trigger a deployment to `PROD`. Verify that Jenkins does not run the `Deploy` to `PROD` in the `deployments` branch



Important

Jenkins requires explicit permission to run deployments in the `PROD` project of OpenShift. You must grant the `edit` role to the Jenkins account in the `PROD` project. To this end, use the `oc policy add-role-to-user` command.

- 10.1. Add the `edit` role to the Jenkins account in the `RHT_OCP4_DEV_USER-deploying-lab-prod` project. Replace `RHT_OCP4_DEV_USER` with the `RHT_OCP4_DEV_USER` provided in the online learning website.

```
[user@host deploying-envs]$ oc policy add-role-to-user \
edit system:serviceaccount:`RHT_OCP4_DEV_USER`-jenkins:jenkins \
-n RHT_OCP4_DEV_USER-deploying-lab-prod
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"
```

- 10.2. Add a `Deploy` to `PROD` stage to your `Jenkinsfile` file, after the `Deploy` to `TEST` stage. The stage only runs if the branch is `main`. The `oc set image deployment` command changes the `home-automation` deployment, by updating the image tag of the `home-automation` container. Note that you must run the command in the `RHT_OCP4_DEV_USER-deploying-lab-prod` project, which is the production namespace.

```

stage('Deploy to PROD') {
    when { branch "main" }

    steps {
        sh """
            oc set image deployment home-automation \
            home-automation=quay.io/${QUAY_USR}/do400-deploying-lab:build-
            ${BUILD_NUMBER} \
            -n RHT_OCP4_DEV_USER-deploying-lab-prod --record
        """
    }
}

```

10.3. Stage and commit the changes.

```

[user@host deploying-lab]$ git add Jenkinsfile
[user@host deploying-lab]$ git commit -m "Added prod deployment"
[deployments f3780c0] Added prod deployment
1 file changed, 12 insertions(+)

```

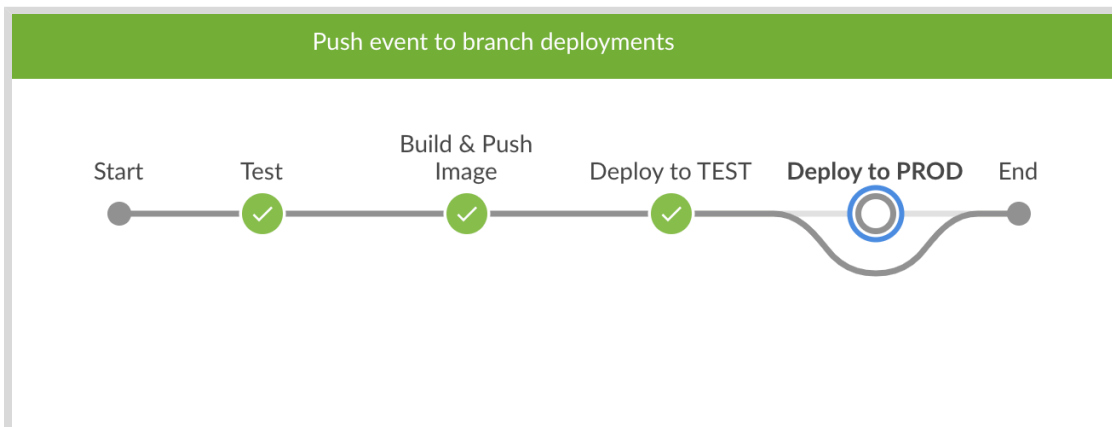
10.4. Push the local commits to the remote GitHub repository.

```

[user@host deploying-lab]$ git push
...output omitted...
To https://github.com/your_github_user/do400-deploying-lab.git
2fce13b..cf70f68 deployments -> deployments

```

10.5. Return to the Blue Ocean browser tab and navigate to the project page. Verify that the pipeline does not execute the **Deploy to PROD** stage in the **deployments** branch.



11. Merge the **deployments** branch into the **main** branch and verify that Jenkins runs the deployment to the **PROD** environment in the **main** branch.



Note

Normally, you should create a pull request to propose your branch for integration into the **main** branch. In this case, we use the `git merge` command for the sake of simplicity.

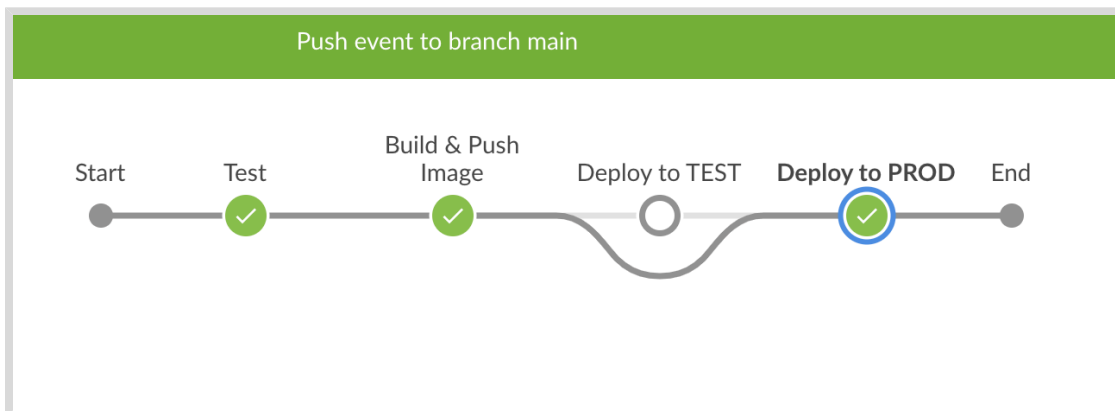
- 11.1. Switch to the `main` branch. Merge the `deployments` branch into the `main` branch.

```
[user@host deploying-lab]$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
[user@host deploying-lab]$ git merge deployments
Updating 8d3d6b2..cf70f68
Fast-forward
 Jenkinsfile | 24 ++++++
 1 file changed, 24 insertions(+)
```

- 11.2. Push the local commits to the `main` branch of the remote GitHub repository.

```
[user@host deploying-lab]$ git push
...output omitted...
To github.com:your_github_user/do400-deploying-lab.git
 8d3d6b2..cf70f68  main -> main
```

- 11.3. Return to the Blue Ocean browser tab. Verify that the pipeline has deployed the application to production. Also note that the pipeline does not run the `Deploy to TEST` stage in the `main` branch.



The logs of the `Deploy to PROD` stage should look like the following:

```
...output omitted...
deployment.apps/home-automation image updated
```

You can also run the `oc rollout history deployment home-automation` command to check that a new revision exists.

```
[user@host deploying-lab]$ oc rollout history deployment home-automation
deployment.apps/home-automation
REVISION  CHANGE-CAUSE
1          <none>
2          oc set image deployment home-automation ...output omitted...
```

12. Clean up. Delete the `TEST` and `PROD` environments, and the Jenkins pipeline project.

- 12.1. Delete the `RHT_OCP4_DEV_USER-deploying-lab-prod` and `RHT_OCP4_DEV_USER-deploying-lab-test` projects from Red Hat OpenShift.

```
[user@host deploying-lab]$ oc delete project RHT_OCP4_DEV_USER-deploying-lab-prod
project.project.openshift.io "your-user-deploying-lab-prod" deleted
[user@host deploying-lab]$ oc delete project RHT_OCP4_DEV_USER-deploying-lab-test
project.project.openshift.io "your-user-deploying-lab-test" deleted
```

- 12.2. Return to the Jenkins home page. Click **do400-home-deploying-lab** to go to the **do400-home-deploying-lab** project detail page. In the left pane, click **Delete Repository** and confirm.

This concludes the lab.

Summary

In this chapter, you learned:

- As part of continuous delivery and continuous deployment, you must be able to automatically deploy your applications.
- You can use containers to package, distribute, and run your application.
- Your pipelines should be able to deploy to different environments.
- You can construct more dynamic pipelines by using environment variables and credentials.
- A deployment strategy is a way of changing or upgrading your application to minimize the impact of those changes.
- Continuous testing is the practice of executing automated tests in a CI pipeline.
- To shorten the feedback loops in CI pipelines, the order in which you execute your tests matters.

Chapter 7

Implementing Pipeline Security and Monitoring Performance

Goal

Manage the security and monitor the performance of pipelines.

Objectives

- Describe and implement DevSecOps principles with Jenkins and OpenShift.
- Scan applications for security vulnerabilities.
- Identify and monitor pipeline performance metrics.
- Configure Jenkins alerts and error notifications.
- Decrease failure rates and rollback failed deployments.

Sections

- Implementing the Basic Principles of DevSecOps (and Guided Exercise)
- Implementing Container Security Scans (and Guided Exercise)
- Defining Performance Metrics (and Guided Exercise)
- Configuring Error Notifications and Alerts (and Guided Exercise)
- Recovering Failures (and Guided Exercise)

Lab

Implementing Pipeline Security and Monitoring

Implementing the Basic Principles of DevSecOps

Objectives

After completing this section, you should be able to describe and implement DevSecOps principles with Jenkins and OpenShift.

Defining DevSecOps

As you have learned previously, DevOps necessitates collaboration and automating the development process to foster an environment of continuous improvement, which leads to better and faster software delivery. Teams achieve this, among other things, by blending together developers and operators in cohesive development teams.

But many DevOps teams lack an important aspect of software development: **Security**.

Security is any measure taken to prevent service disruption and data leakage or loss.

Development Security Operations (DevSecOps) aims at integrating security throughout the whole software development process. An Agile team can accomplish this by changing their mindset to understand all the security implications related to the software they are building. In the past, security was verified by a specific team usually towards the end of the software life cycle. With DevSecOps, security is a shared responsibility, integrated into the whole life cycle through automation.

Introducing the DevSecOps Process

This process of securing both the development process and the application code consists of several parts:

1. Secure development infrastructure
 - Automate and standardize the development environment.
 - Minimize the privileges to access all services.
 - Isolate containers to prevent attacks.
 - Use encrypted communication between infrastructure services.
 - Centralize access control.
 - Automate security updates.
2. Security concerns in software development
 - Use encrypted communication between application services.
 - Secure APIs.
3. Add security concerns in testing
 - Add automatic security scans in Continuous Integration.

- Automate input validation acceptance tests.

To support DevOps continuous improvement in the code and the infrastructure, teams engaging in DevOps development must improve their knowledge and skills continuously to improve their processes and delivery. The same principle applies to DevSecOps, which requires this continuous refinement to provide the best security possible for both the development process and the application itself.

Practicing DevSecOps also requires proactively testing the application's security and encourages creating **Red Teams**: teams dedicated to constantly test the application for security vulnerabilities.

Creating Jenkins Credentials

As we have covered earlier, a CI/CD pipeline such as Jenkins, is a key tool to successfully apply Agile principles in DevOps. Because Jenkins is such a central piece, in a DevSecOps development team it is imperative that Jenkins connects security to all the external services it might need.

To avoid hard coding the user name and password of these external services in every pipeline, Jenkins provides the concept of Credentials.

A *Credential* consists of a shared mechanism to authenticate in a particular service. This mechanism can be an API token or an SSH key pair, which a pipeline must use to connect to services that require authentication to operate with it.

Having this authentication mechanism outside the pipeline achieves several purposes:

1. Keeps the connection to the external service secure.
2. Separation of external services access permissions from authentication mechanism ownership or access.
3. Shares the authentication credentials among several pipelines.
4. Eases the authentication mechanism update or refresh, updating all occurrences at once.

To ensure that only the allowed users can use a particular credential, Jenkins can define them in several locations:

- Global credentials are accessible to all users in all the pipelines.
- Per-project credentials that are accessible in the project pipelines.
- Per-user credentials that the user can access in the pipelines they have permissions to.

You can administer credentials in the **Manage credentials** section of the Jenkins configuration.

Adding GitHub Authentication and Authorization

CI/CD pipelines need access to the source code repositories, and these repositories need to connect to the pipelines for notification purposes. From the DevSecOps perspective, this access must be secured by using *Access Tokens*.

Access tokens are a representation of a user's security credentials and privileges. A token representation can take multiple forms, but the most common one is a 40 character string.

GitHub allows the creation of multiple access tokens that you can use in other services, such as Jenkins, so they can authenticate as your user in GitHub. Access tokens allows you to avoid using

your user name and password in external services, and also allows you to control the access that third party services have to your GitHub account.

To be able to make this fine-grained control, you must use one access token for each different service that you wish to allow access to your account.

To create a GitHub access token click **Settings** and then click **Developer settings** → **Personal access tokens** in the left pane, and then click **Generate new token** in the right pane.

A second level of control in GitHub is the permission levels inside an organization. Users in GitHub can belong to none, one, or more organizations; and this allows organization owners to control what repositories each user can access and what permissions said user has in each project.

Declaring OpenShift Authentication and Authorization

Just as GitHub and Jenkins allow creating access tokens and credentials, *Red Hat OpenShift Container Platform* (RHOCP) allows the creation of **Secrets**. Secrets are the equivalent of Jenkins' Credentials, use them to store shared authentication mechanisms that the RHOCP cluster must use to connect to external services such as GitHub.

When creating a project that needs to replicate a secured repository, you must create a Secret containing a GitHub access token. For example, to create a secret from a private SSH key use:

```
[user@host ~]$ oc create secret generic sshsecret \
--from-file=ssh-privatekey=$ HOME/.ssh/id_rsa \
-n your-application-project
```

In the same way RHOCP connects to external services, third party systems might need access to secured RHOCP endpoints. To avoid having these third party services use a user's account to connect, RHOCP can create Service Accounts that are not tied to a particular user.

You can create service accounts global to the system or tied to a particular project, depending on you isolation level needs.

For example, to allow the jenkins service deployment access to the jenkins-project project in the your-application-project RHOCP namespace you must use:

```
[user@host ~]$ oc policy add-role-to-user \
edit system:serviceaccount:jenkins-p roject:jenkins \
-n your-application-project
```



References

devsecops.org

<https://www.devsecops.org/>

What is DevSecOps?

<https://www.redhat.com/en/topics/devops/what-is-devsecops>

Using Jenkins Credentials

<https://www.jenkins.io/doc/book/using/using-credentials/>

Creating a token

<https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token#creating-a-token>

► Guided Exercise

Implementing the Basic Principles of DevSecOps

In this exercise you will set up different mechanisms to secure automated CI pipelines.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\\) to introduce line breaks in these commands.

Outcomes

You should be able to build automated Jenkins pipelines with the following capabilities:

- Clone code from private repositories
- Deploy applications in OpenShift from private repositories
- Interact with external services using credentials within a `Jenkinsfile` file
- Push images to external private registries

Before You Begin

To perform this exercise, ensure you have:

- Your `D0400`-apps fork cloned in your workspace
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- A GitHub account
- A Quay.io account
- The OpenShift CLI



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Create a private repository in GitHub.

- 1.1. In GitHub, create a new **private** empty repository called `YOUR_GITHUB_USER/do400-greeting-devsecops`.
- 1.2. From your workspace folder, navigate to the `D0400-apps` application folder and checkout the `main` branch to ensure that you start the exercise from a clean state.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
...output omitted...
```

- 1.3. Copy the contents of the `greeting-cd-pipeline` folder to a new `greeting-devsecops` folder in your workspace. Navigate to the newly created content.

```
[user@host D0400-apps]$ cp -Rv greeting-cd-pipeline ~/D0400/greeting-devsecops
...output omitted...
[user@host D0400-apps]$ cd ~/D0400/greeting-devsecops
[user@host greeting-devsecops]$
```



Important

Windows users must replace the preceding `cp` command in PowerShell as follows:

```
PS C:\Users\user\D0400\D0400-apps> Copy-Item -Path greeting-cd-pipeline `
>> -Destination ~\D0400\greeting-devsecops -Recurse
```

- 1.4. Initialize the repository and push it to the GitHub remote. Enter your GitHub user name and personal access token when prompted.

```
[user@host greeting-devsecops]$ git init
Initialized empty Git repository in ...output omitted.../greeting-devsecops/.git/
[user@host greeting-devsecops]$ git add -A
[user@host greeting-devsecops]$ git commit -m "first commit"
[master (root-commit) e2cdc83] first commit
 5 files changed, 65 insertions(+)
...output omitted...
[user@host greeting-devsecops]$ git branch -M main
[user@host greeting-devsecops]$ git remote add origin \
https://github.com/YOUR_GITHUB_USER/do400-greeting-devsecops.git
[user@host greeting-devsecops]$ git push -u origin main
Username for 'https://github.com': YOUR_GITHUB_USER
Password for 'https://your-github-user@github.com':
...output omitted...
* [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

► 2. Create a GitHub access token.

- 2.1. Open a browser tab and navigate to <https://github.com/settings/tokens>.
- 2.2. Click **Generate new token**. Enter your GitHub password when prompted. Create a new token with the `repo` scope selected. Enter `do400-devsecops` for the Note field. After generating the token, copy the token value.

**Note**

The repo scope allows access to private repositories.

**Warning**

Be sure to copy the generated access token and store it somewhere. You will use it in several steps of this exercise.

- ▶ **3.** Create a new Jenkins pipeline with access to your private repository.
 - 3.1. Open the Jenkins dashboard in a web browser. Click **New Item**.
 - 3.2. Enter `greeting-devsecops` for the name, select **MultiBranch Pipeline** for the type, and click **OK**.
 - 3.3. In the **Branch Sources** section, click **Add source** and select **Git**.
 - 3.4. Enter `https://github.com/YOUR_GITHUB_USER/do400-greeting-devsecops.git` as **Project Repository**.
 - 3.5. In **Credentials**, click **Add** and select `greeting-devsecops`. The **Folder Credentials Provider: greeting-devsecops** window opens.
 - 3.6. In the **Username** field, enter your GitHub user name. In the **Password** field, enter the token you created before. Enter `github-private` for the ID. Type `greeting-devsecops` in the **Description** field to differentiate from other tokens you may have. Click **Add**.

**Note**

GitHub is deprecating password-based authentication for all Git operations in favour of personal tokens, GitHub App installation tokens or OAuth flow: <https://github.blog/2020-12-15-token-authentication-requirements-for-git-operations/>

Despite Jenkins labels the field as **Password**, in this exercise you are using a personal token, hence this deprecation does not impact the exercise.

For an enterprise-grade project, the recommended approach is using SSH-based authentication and a private key infrastructure.

- 3.7. Select `YOUR_GITHUB_USER/***** (greeting-devsecops)` as the credential.
- 3.8. Click **Save**. Jenkins saves the pipeline and starts indexing the repository branches.
- 3.9. Navigate to the `greeting-devsecops` main page and click **main** in the branches list. Check that the pipeline build succeeds.
- ▶ **4.** Create another pipeline. Verify that you cannot use the same GitHub credentials in another pipeline.
 - 4.1. From the Jenkins main page, click **New Item**.
 - 4.2. Enter `greeting-devsecops-2` for the name, select **MultiBranch Pipeline** for the type, and click **OK**.

- 4.3. In the **Branch Sources** section, click **Add source** and select **Git**.
- 4.4. Enter `https://github.com/YOUR_GITHUB_USER/do400-greeting-devsecops.git` as **Project Repository**. Note the **Credentials** selector does not show the credential that you configured for the previous pipeline.
- 4.5. Click **Save**. Jenkins fails to scan the repository branches due to an authentication error in GitHub.
- ▶ 5. Create a global credential and use it in the `greeting-devsecops-2` pipeline.
 - 5.1. From the Jenkins main page, click **Manage Jenkins** in the left pane, and then click **Manage Credentials** in the right pane.
 - 5.2. In the **Stores scoped to Jenkins** section of the right pane, click **Jenkins**.
 - 5.3. Click **Global credentials (unrestricted)** in the right pane, and then click **Add Credentials** in the left pane.
 - 5.4. To create a new credential, use the your GitHub user name and the same GitHub access token for the password. Enter `github-global` for the ID and click **OK** to create the credential.
 - 5.5. Navigate to the `greeting-devsecops-2` pipeline details page and click **Configure**. In the **Branch Sources** section, click the **Credentials** selector and select the global credential that you just created.
 - 5.6. Click **Save**. Jenkins successfully scans the repository branches and runs the pipeline for the `main` branch. Wait for the pipeline to finish and verify that the pipeline succeeds.
- ▶ 6. Deploy the application to OpenShift by using a source clone secret. The source clone secret allows OpenShift to securely clone the code from your private repository.
 - 6.1. In the command-line terminal, login to OpenShift and create a new project called `RHT_OCP4_DEV_USER-devsecops`.

```
[user@host greeting-devsecops]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
[user@host deploying-envs]$ oc new-project RHT_OCP4_DEV_USER-devsecops
Now using project "youruser-devsecops" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 6.2. Create a new source clone secret by using your GitHub access token. Replace `YOUR_GITHUB_ACCESS_TOKEN` with the GitHub access token that you created before.

```
[user@host greeting-devsecops]$ oc create secret generic github-secret-token \
--from-literal=password=YOUR_GITHUB_ACCESS_TOKEN \
--type=kubernetes.io/basic-auth
secret/github-secret-token created
```

- 6.3. Create a new application. Use the `--source-secret` parameter to specify the `github-secret-token` secret. Enter your GitHub user name and personal access token when prompted.

```
[user@host greeting-devsecops]$ oc new-app --name greeting-devsecops \
https://github.com/YOUR_GITHUB_USER/do400-greeting-devsecops.git \
--strategy=docker \
--source-secret=github-secret-token
Username for 'https://github.com': YOUR_GITHUB_USER
Password for 'https://your-github-user@github.com':
warning: Cannot check if git requires authentication.
...output omitted...
--> Success

Build scheduled, use 'oc logs -f bc/greeting-devsecops' to track its progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/greeting-devsecops'
Run 'oc status' to view your app.
```

You might need to specify your GitHub user and password once or twice, depending on your local git credentials cache configuration.

- 6.4. Expose the app.

```
[user@host greeting-devsecops]$ oc expose svc/greeting-devsecops
route.route.openshift.io/greeting-devsecops exposed
```

- 6.5. Get the URL of the application.

```
[user@host greeting-devsecops]$ oc get route greeting-devsecops \
-o jsonpath="{ 'http://' }{ .spec.host }{ '\n' }"
greeting-devsecops-your-user-devsecops.apps.cluster.example.com
```

Open a new browser tab and navigate to the URL of the application. Verify that the application returns the `Hello student` response. You might need to wait a few seconds until the application is completely deployed.

► 7. Allow Jenkins to run deployments on OpenShift.

- 7.1. Open the `Jenkinsfile` file. Set the `APP_NAMESPACE` variable to `RHT_OCP4_DEV_USER-devsecops` and add the `Deploy` stage. Replace `RHT_OCP4_DEV_USER` with your lab user. The file should display as follows:

```
pipeline {
  agent { label 'nodejs' }

  environment { APP_NAMESPACE = 'RHT_OCP4_DEV_USER-devsecops' }

  stages{

    stage('Test') {
      steps{
        sh "node test.js"
      }
    }
  }
}
```

```

    }

    stage('Deploy') {
        steps {
            sh '''
                oc start-build greeting-devsecops \
                --follow --wait -n ${APP_NAMESPACE}
            '''
        }
    }
}

```

7.2. Save the file, commit the changes and push the branch to your private repository.

```

[user@host greeting-devsecops]$ git commit Jenkinsfile -m "activate deployment"
[main fdf0b72] activate deployment
1 file changed, 1 insertion(+), 3 deletions(-)
[user@host greeting-devsecops]$ git push origin main
...output omitted...
1e64f78..fdf0b72  main -> main

```

7.3. Add the view role to the jenkins service account.

```

[user@host greeting-devsecops]$ oc policy add-role-to-user view \
system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins \
-n RHT_OCP4_DEV_USER-devsecops
clusterrole.rbac.authorization.k8s.io/view added: "system:serviceaccount:your-
user-jenkins:jenkins"

```

Note that the role binding is applied to the *RHT_OCP4_DEV_USER-devsecops* project by using the *jenkins* account, which is taken from the *RHT_OCP4_DEV_USER-jenkins* project. This allows pods running in the *RHT_OCP4_DEV_USER-jenkins* to interact with the *RHT_OCP4_DEV_USER-devsecops* project.

7.4. In Jenkins, navigate to the *greeting-devsecops* pipeline page, click the *main* branch, and run the pipeline for this branch. The pipeline fails due the following error:

```

+ oc start-build greeting-devsecops --follow --wait
Error from server (Forbidden): buildconfigs.build.openshift.io "greeting-
devsecops" is forbidden: User "system:serviceaccount:your-user-jenkins:jenkins"
cannot create resource "buildconfigs/instantiate" in API group
"build.openshift.io" in the namespace "your-user-devsecops"

```

This is because the *view* role does not allow the creation of resources in the *RHT_OCP4_DEV_USER-devsecops* project.

7.5. Add the edit role to the jenkins service account.

```
[user@host greeting-devsecops]$ oc policy add-role-to-user edit \
system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins \
-n RHT_OCP4_DEV_USER-devsecops
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:your-
user-jenkins:jenkins"
```

- 7.6. In Jenkins, run the `greeting-devsecops` pipeline again. The `Deploy` stage succeeds, building a new image and updating the deployment. Check that the deployment has created a new `ReplicaSet` to update the pods with the new image.

```
[user@host greeting-devsecops]$ oc get replicaset
NAME                                DESIRED    CURRENT    READY    AGE
...output omitted...
greeting-devsecops-5b77569699      1          1          1        28s
greeting-devsecops-66c9b54768      0          0          0        25m
...output omitted...
```

- 8. Use a credential in a pipeline to perform actions that require authentication.

- 8.1. Add the `Push to Quay` stage. Open the `Jenkinsfile` file in your editor and add the `Push to Quay` stage after the `Deploy` stage. The stage should display as follows:

```
stage('Push to Quay') {
  steps {
    sh '''
      oc start-build greeting-devsecops-quay \
      --follow --wait -n ${APP_NAMESPACE}
    '''
  }
}
```

- 8.2. Add a post-condition hook after the stages section. The hook creates an issue in GitHub if the pipeline fails. Use the `github-global` credential that you created in a previous step.

```

pipeline {
    ...output omitted...
    stages {
        ...output omitted...
    }

    post {
        failure {
            withCredentials([usernamePassword(
                credentialsId: 'github-global',
                usernameVariable: 'USERNAME',
                passwordVariable: 'PASSWORD'
            )]) {
                sh """
                    curl -X POST \
                    -H 'Authorization: token $PASSWORD' \
                    'https://api.github.com/repos/$USERNAME/do400-greeting-
devsecops/issues' \
                    -d '{"title": "CI build $BUILD_NUMBER", "body": "Pipeline
build $BUILD_NUMBER has failed"}'
                """
            }
        }
    }
}

```

8.3. Save the file, commit the changes and push the branch to your private repository.

```

[user@host greeting-devsecops]$ git commit Jenkinsfile -m "activate push and hook"
[main 100e40b] activate push and hook
1 file changed, 21 insertions(+)
[user@host greeting-devsecops]$ git push origin main
...output omitted...
fdf0b72..100e40b  main -> main

```

8.4. In Jenkins, run the `greeting-devsecops` pipeline again. The pipeline fails because you have not created the `greeting-devsecops-quay` build in OpenShift yet. Jenkins triggers the hook, which creates a new issue in your GitHub repository.

8.5. In a new browser tab, navigate to https://github.com/YOUR_GITHUB_USER/do400-greeting-devsecops/issues. Check that the new issue has been created in GitHub.

- ▶ 9. Fix the pipeline. Create the `greeting-devsecops-quay` build in OpenShift and authorize the build to push images to Quay.io.
 - 9.1. Navigate to <https://quay.io> to download your credentials file. In your account settings, click **Generate Encrypted Password**. From the **Docker Configuration** option, click **Download yourquayuser-auth.json** to download the credentials file.
 - 9.2. Create a secret by using the credentials file that you just downloaded. Specify the file path of the `yourquayuser-auth.json` file with the `from-file` parameter.

```
[user@host greeting-devsecops]$ oc create secret generic \
quay-encrypted-auth \
--from-file=.dockerconfigjson=/absolute/path/to/yourquayuser-auth.json \
--type=kubernetes.io/dockerconfigjson \
-n RHT_OCP4_DEV_USER-devsecops
secret/quay-encrypted-auth created
```

**Warning**

The `oc create secret` command does not support relative paths. Use an absolute path for the `--from-file` parameter.

- 9.3. Configure a `BuildConfig` resource to build and push your application image to Quay.io. The build must use two secrets. The `--source-secret` parameter provides the build with the credentials to clone the code from the private repository. The `--push-secret` parameter provides the build with the credentials to push the built image to Quay.io.

```
[user@host greeting-devsecops]$ oc new-build \
https://github.com/YOUR_GITHUB_USER/do400-greeting-devsecops.git \
--name=greeting-devsecops-quay \
--strategy=docker \
--source-secret=github-secret-token \
--to-docker=true \
--to=quay.io/YOUR_QUAY_USER/greeting-devsecops \
--push-secret=quay-encrypted-auth \
-n RHT_OCP4_DEV_USER-devsecops
Username for 'https://github.com': YOUR_GITHUB_USER
Password for 'https://your-github-user@github.com':
warning: Cannot check if git requires authentication.
...output omitted...
--> Creating resources with label build=greeting-devsecops-quay ...
    buildconfig.build.openshift.io "greeting-devsecops-quay" created
--> Success
```

You might need to specify your GitHub user and personal access token once or twice, depending on your local git credentials cache configuration.

- 9.4. Run the `greeting-devsecops` pipeline again. The pipeline should pass. Verify that the new image has been pushed to your Quay.io account.

► **10.** Clean up.

- 10.1. Delete the `RHT_OCP4_DEV_USER-devsecops` OpenShift project and navigate back to your workspace folder.

```
[user@host greeting-devsecops]$ oc delete project RHT_OCP4_DEV_USER-devsecops
[user@host greeting-devsecops]$ cd ..
[user@host D0400]$
```

- 10.2. In Jenkins, delete the `greeting-devsecops` and `greeting-devsecops-2` pipelines, and the `github-global` credential.

10.3. In Quay.io, delete the `greeting-devsecops` image from your account.

This concludes the guided exercise.

Implementing Container Security Scans

Objectives

After completing this section, you should be able to scan applications for security vulnerabilities.

Defining Container Security

Security breaches are a global, regular occurrence. No matter the size of your business, you can be the target of an attack. The most important consequences that a security breach can lead to are legal consequences, financial loss, reputation damage, operational downtimes, and loss of sensitive data.

In cloud-native developments, you can reduce the risk of security breaches by applying *container security* practices. Container security is the protection of the integrity of containers. Everything from your application to the infrastructure you use for deployments must be secured and protected. If a container is compromised or accessible by bad agents, then the attacker can potentially affect other containers or systems unless you have appropriate counter measures in place.



Note

A best practice for application security is to integrate and automate the security tests in your CI/CD pipelines. By building security into your pipelines, you can make sure your containers are trusted and reliable.

Building Secure Container Images

Container security begins with a base container image and continues with the rest of the image layers. This base image is the starting point from which you create derivative images by adding layers for your application and its dependencies. Usually, the base container image contains the operating system, and the required runtime for your application. For example, a container image for a Node.js application could use a *RHEL* image as the base layer. The rest of the layers would include the Node.js runtime, the application code, and the start command.

As you can see, the base image plays an essential role in the security of your containers. This is the reason why you must **use container images from trusted sources** when possible.

With Red Hat OpenShift you can automate the entire image build process. You can configure automated actions to execute when you update your container images. For example, you can automatically rebuild all your container images when you update your container base image.



Note

Container images are immutable, an image can only be fixed by a new version of that image.

The Red Hat Ecosystem Catalog

Using a container image from a public registry as your base container image is often the fastest and simplest solution, but in many instances you might not know the reliability or safety of that base image. Public registry images can pose lots of questions that might not have clear answers. Is it privileged or unprivileged? Are you sure that all the layers are up to date? How quickly is the image updated in the case of a security flaw?

As an alternative, Red Hat provides a catalog of certified container images. Those container images are built from base images that have been vetted by the Red Hat internal security team, and hardened against security flaws.

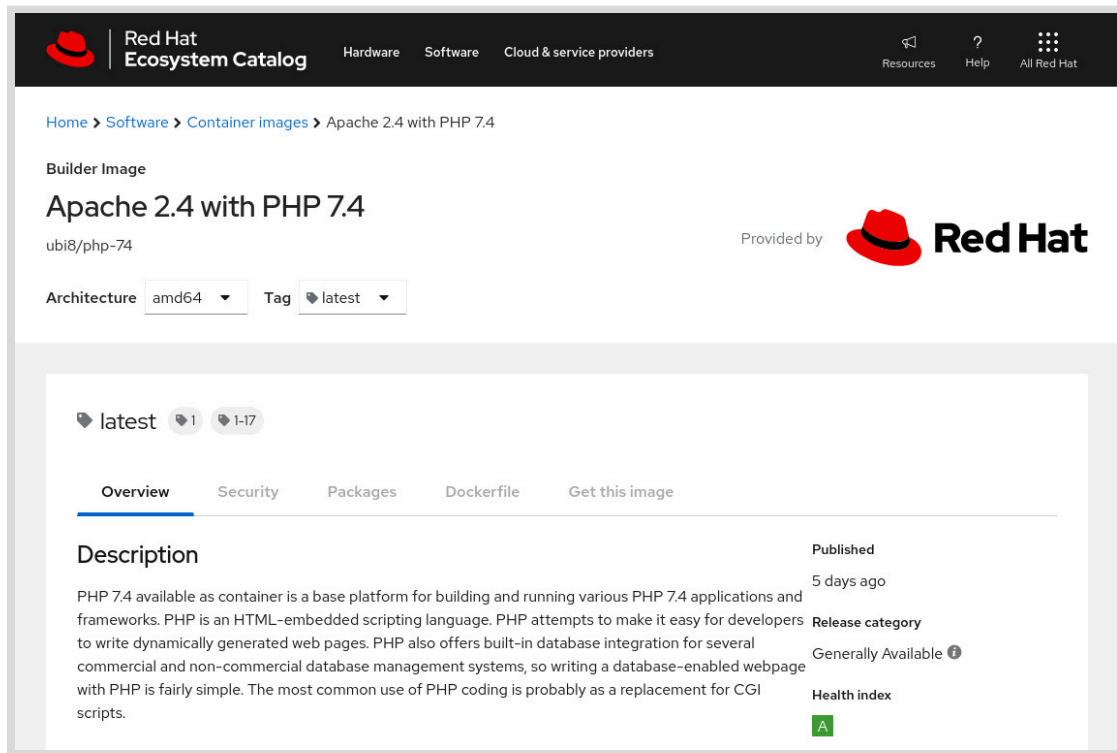


Figure 7.1: Certified container image

Red Hat also rates the catalog images based on published non-applied security updates, and how long the software in the container images is exposed to those flaws. This rating system is called the *Container Health Index*, and each image has a grade according to a scale from A to F.

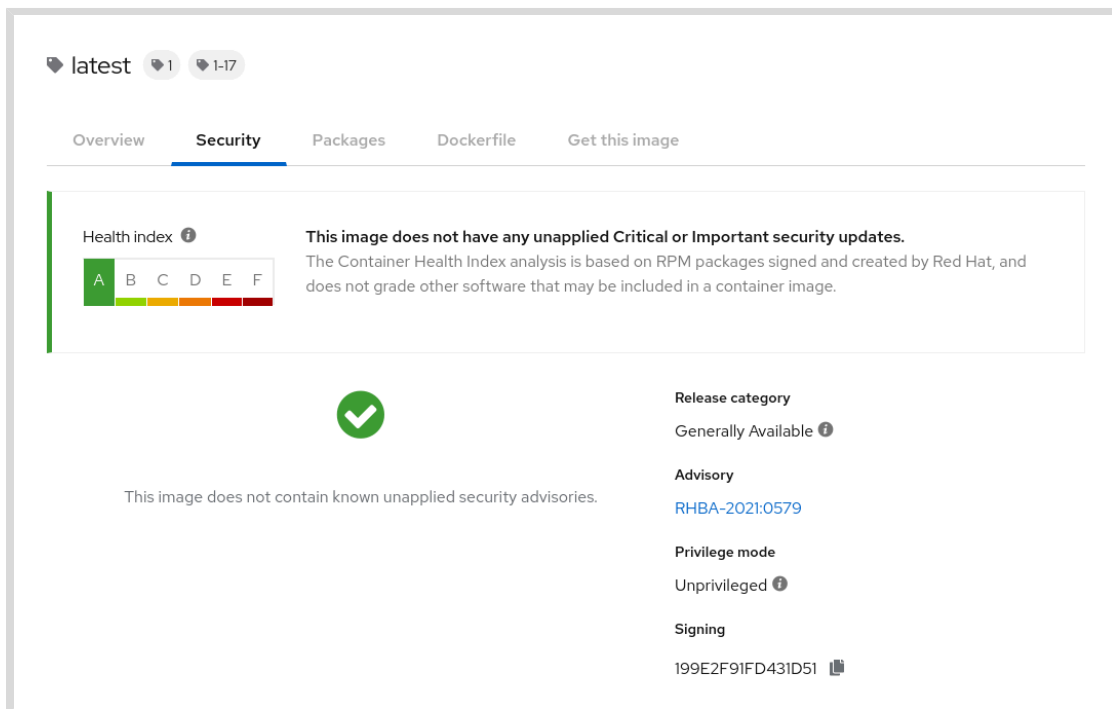


Figure 7.2: Security information about a certified container image

As new advisories become publicly available, Red Hat scans all the images available in its ecosystem inventory to see if any are affected. If an affected image is found, then the image grade is updated.

Managing Access to Container Images

Another layer of container security is the access management and promotion of all container images your team uses. That means protecting the images you use as well as the ones you build.

By using a private registry you not only can control who has access to the images, you increase the traceability of your container images by assigning metadata to them. For example, you can tag your container images with the specific versions of the dependencies used by your application.

You can use the internal and private registry available in Red Hat OpenShift, or use an external one such as Quay.io.



Note

Quay.io and Red Hat Quay automatically scans your containers for vulnerabilities. They give you complete visibility into known issues and how to fix them.

Scanning Container Images

Continued security relies on scanning container images for vulnerabilities and by having an efficient way to replace vulnerable images. You can accomplish those goals with the integration of static analysis tools in your CI/CD pipelines.

Static analysis tools examine all the container image layer to find known vulnerabilities. Some of the most known tools for scanning your container images are Trivy, Clair, OpenSCAP, and Anchore.

Integrating Security Scanners in Pipelines

With the help of CI/CD pipelines, you have an automated and reproducible way of building your application. Depending on the tool you select for scanning your containers, the integration of a security scan can be as simple as executing a Kubernetes job, which performs the scan, and extracts the result.

For example, you can have a Kubernetes Job definition in your application repository. The following snippet shows a job that uses Trivy for the image scan.

```
kind: Job
apiVersion: batch/v1
metadata:
  name: app-scanned-with-trivy
spec:
  backoffLimit: 0
  template:
    metadata:
      name: app-scanned-with-trivy
    spec:
      containers:
        - name: app-scanned-with-trivy
          image: image-registry.openshift-image-registry.svc:5000/project/app 1
          command: [ "/bin/sh" ]
          args:
            - -c
            - >-
              cd /tmp &&
              curl -sL https://trivy/release.tar.gz -o - | tar -zxf - && 2
              ./trivy fs --exit-code 1 --severity HIGH,CRITICAL --no-progress / 3
          restartPolicy: Never
```

- ¹ The job uses your application container image to execute commands.
- ² The command to execute in the container first installs Trivy.
- ³ Trivy scans all the content of the container.

Given that you have the Job definition available in your repository, you can integrate the Job execution in your pipeline. The following snippet shows how you can integrate the scanning job with your Jenkins pipeline.

```
...output omitted...
stage('Security Scan') {
  steps {
    sh 'oc create -f security-scan-job.yml' 1
  }
}
...output omitted...
```

- ¹ Creates a job in Kubernetes to scan your application container for vulnerabilities.

In your pipelines, you must stop the deployment of any container image flagged with security flaws. At the moment in which there is a patch to solve the vulnerability, you must rebuild your application images and deploy them.



References

Certified Container Images

<https://catalog.redhat.com/software/containers/explore>

Container Health Index grades as used inside the Red Hat Container Catalog

<https://access.redhat.com/articles/2803031>

Clair

<https://github.com/quay/clair>

► Guided Exercise

Implementing Container Security Scans

In this exercise you will create a pipeline that scans for vulnerabilities before deploying an application.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to scan for vulnerabilities in your application containers.

Before You Begin

To perform this exercise, ensure you have:

- Your D0400-apps fork cloned in your workspace
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- The OpenShift CLI



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the Shopping Cart application.
 - 1.1. From your workspace folder, navigate to the D0400-apps/shopping-cart application folder. Checkout the solutions branch of the D0400-apps repository.

```
[user@host D0400]$ cd D0400-apps/shopping-cart
[user@host shopping-cart]$ git checkout solutions
...output omitted...
```

- 1.2. Create a new branch named security-scans to save any changes you make during this exercise, and push the branch to the remote repository.

```
[user@host shopping-cart]$ git checkout -b security-scans
Switched to a new branch 'security-scans'
[user@host shopping-cart]$ git push origin HEAD
...output omitted...
* [new branch]      HEAD -> security-scans
```

► 2. Create an environment in OpenShift to deploy the application.

- 2.1. Log in to OpenShift with the credentials provided in the **Lab Environment** tab of the Red Hat online learning website. You can also copy the login command from the OpenShift web console.

```
[user@host shopping-cart]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
```

- 2.2. Create an OpenShift project to host the Shopping Cart application.

```
[user@host shopping-cart]$ oc new-project RHT_OCP4_DEV_USER-security-scans
Now using project "youruser-security-scans" on server "https://
api.cluster.domain.example.com:6443".
...output omitted...
```

- 2.3. Add the edit role to the Jenkins account in the *RHT_OCP4_DEV_USER-security-scans* project.

```
[user@host shopping-cart]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-
jenkins:jenkins"
```

Notice that you must replace the *RHT_OCP4_DEV_USER* string with your OpenShift developer user.

► 3. Deploy the Shopping Cart application to OpenShift and check that the application is running correctly.

- 3.1. Process the *kubefiles/image-builder-template.yml* file to create all the required Kubernetes resources for the application to work.

```
[user@host shopping-cart]$ oc process -f kubefiles/image-builder-template.yml \
-p GIT_REPOSITORY=https://github.com/YOUR_GITHUB_USER/DO400-apps \
-p GIT_REF=security-scans \
-p PROJECT_NAME=RHT_OCP4_DEV_USER-security-scans \
| oc apply -f -
imagestream.image.openshift.io/openjdk-11 created
imagestream.image.openshift.io/security-scans created
buildconfig.build.openshift.io/security-scans created
deployment.apps/security-scans created
service/security-scans created
route.route.openshift.io/security-scans created
```

The preceding command creates all the resources required to build and deploy the application. The template has no automatic triggers for the build and deploy phases. This makes it possible to run them in different stages of the pipeline.

- 3.2. Use the `oc start-build` command to generate the application image.

```
[user@host shopping-cart]$ oc start-build security-scans \
--follow --wait
...output omitted...
Push successful
```

- 3.3. Use the `oc rollout status` command to wait until the Deployment pulls the application image. You might need to wait a few minutes until the deployment is rolled out.

```
[user@host shopping-cart]$ oc rollout status deployment security-scans --watch
Waiting for deployment "security-scans" rollout to finish: 0 of 1 updated replicas
are available...
deployment "security-scans" successfully rolled out
```

- 3.4. Get the application route.

```
[user@host shopping-cart]$ oc get route security-scans \
-o jsonpath='{\"http://\"}{.spec.host}{\"/cart\\n\"}'
http://security-scans-your-user-security-scans.apps.cluster.example.com/cart
```

- 3.5. Open a browser and navigate to the URL that you just retrieved. Notice that the Quarkus application returns a JSON response composed of the `products` key with an empty list as value, and the `totalItems` key with 0 as value.

- 4. Create a declarative pipeline composed of different stages executed sequentially. The stages for the pipeline are the following:

- Run the tests.
- Build the application image.
- Deploy the application to OpenShift.

- 4.1. Create a file named `Jenkinsfile` in the root of the application folder to store the pipeline. The file contents should look similar to this:


```

pipeline {
  agent { node { label 'maven' } }
  environment { APP_NAMESPACE = 'RHT_OCP4_DEV_USER-security-scans' }
  stages {
    stage('Test') {
      steps {
        dir('shopping-cart') {
          sh './mvnw clean test'
        }
      }
    }
    stage('Image Build') {
      steps {
        dir('shopping-cart') {
          sh """
            oc start-build security-scans \
              --follow --wait -n ${APP_NAMESPACE}
          """
        }
      }
    }
    stage('Deploy') {
      steps {
        dir('shopping-cart') {
          sh """
            oc patch deployment security-scans \
              -p '{"spec": {"template": {"metadata": {"labels":
{"build": "build-${BUILD_NUMBER}"}}}}' \
              -n ${APP_NAMESPACE}
          """
        }
      }
    }
  }
}

```

4.2. Stage and commit the Jenkinsfile file to the local repository.

```

[user@host shopping-cart]$ git add Jenkinsfile
[user@host shopping-cart]$ git commit -m "Added Jenkins integration"
[main eb6ad6c] Added Jenkins integration
1 file changed, 27 insertions(+)
create mode 100644 Jenkinsfile

```

4.3. Push the local commits to the remote repository with the `git push` command.

```

[user@host shopping-cart]$ git push origin HEAD
...output omitted...
19be08f..5adf7f9 HEAD -> security-scans

```

► 5. Create a Jenkins project to run the pipeline.

- 5.1. Open a new tab in the browser and navigate to the URL of the Jenkins server that you installed in *Guided Exercise: Configuring a Developer Environment*. If prompted, log in into Jenkins with your OpenShift credentials.
- 5.2. In the left pane, click **New Item**.
- 5.3. Assign the name `do400-security-scans` to the project, then click **Pipeline**, and finally click **OK**.
- 5.4. Select the **Discard old builds** check box, and then type 3 in the **Max # of builds to keep** field.
- 5.5. Click the **Pipeline** tab. In the **Pipeline** area, from the **Definition** list, select **Pipeline script from SCM**.
- 5.6. From the **SCM** list select **Git**. In the **Repository URL** type the URL of the GitHub repository created in the first steps.
The URL of the repository is similar to `https://github.com/YOUR_GITHUB_USER/DO400-apps`.
- 5.7. In the **Branch Specifier** field, type `*/security-scans`.
- 5.8. In the **Script Path** field, type `shopping-cart/Jenkinsfile`.
- 5.9. Click **Save** to save the pipeline and go to the pipeline details page.
- 5.10. From the left side navigation, click **Build Now** to schedule a pipeline run. Wait for the build to finish and verify that it is successful.

► 6. Create a Job resource to scan the application container for vulnerabilities.

- 6.1. Open the `kubefiles/security-scan-template.yml` file and examine the code. The template creates a Job that uses the application image, and injects Trivy in the container to run a security scan. Trivy is an open source tool that detects vulnerabilities of operating systems and application dependencies.



Note

Security experts find new vulnerabilities every day. This exercise only looks for one of them, CVE-2020-14583, to ensure that the result of the exercise remains deterministic.

In a real world scenario, you should check for all the vulnerabilities.

- 6.2. Run the security scan.

```
[user@host shopping-cart]$ oc process -f \
kubefiles/security-scan-template.yml \
-p PROJECT_NAME=RHT_OCP4_DEV_USER-security-scans \
-p CVE_CODE=CVE-2020-14583 \
| oc replace --force -f -
job.batch/security-scans-trivy replaced
```

The preceding command processes the template and creates a job to scan the application container for vulnerabilities. The `oc replace` command removes any previous job and recreates the same job. This command allows you to run the same job multiple times. The `--force` option removes any previous job immediately.

6.3. Examine the job logs to verify that the image contains vulnerabilities.

```
[user@host shopping-cart]$ oc logs -f job/security-scans-trivy
| java-11-openjdk          | CVE-2020-14583 ...output omitted...
| java-11-openjdk-devel    | CVE-2020-14583 ...output omitted...
| java-11-openjdk-headless | CVE-2020-14583 ...output omitted...
```



Note We are using an old base image in this guided exercise, and the number of vulnerabilities might differ from the example.

- ▶ 7. Add a new stage in the pipeline to check for vulnerabilities before the application is deployed.

7.1. Open the `Jenkinsfile` file and add a stage named `Security Scan` between the `Image Build` and `Deploy` stages. The file contents should look similar to this:

```
...output omitted...
stage('Image Build') {
    ...output omitted...
}
stage('Security Scan') {
    steps {
        dir('shopping-cart') {
            sh """
                oc process -f kubefiles/security-scan-template.yml \
                -n ${APP_NAMESPACE} \
                -p PROJECT_NAME=${APP_NAMESPACE} \
                -p CVE_CODE=CVE-2020-14583 \
                | oc replace --force -n ${APP_NAMESPACE} -f -
            """
            sh "! ../tools/check-job-state.sh security-scans-trivy
            ${APP_NAMESPACE}"
        }
    }
}
stage('Deploy') {
    ...output omitted...
```

The stage processes the `security-scan-template.yml` file to run the scan job, and uses the `tools/check-job-state.sh` script to check if a given vulnerability exists in the `security-scans-trivy` job logs. The `tools/check-job-state.sh` script will fail if the containers have the `CVE-2020-14583` vulnerability.

7.2. Stage and commit the `Jenkinsfile` file to the local repository.

```
[user@host shopping-cart]$ git add Jenkinsfile
[user@host shopping-cart]$ git commit -m "Added Scan stage"
[security-scans e3d16f6] Added Scan stage
1 file changed, 13 insertions(+)
```

7.3. Push the local commits to the remote repository with the `git push` command.

```
[user@host shopping-cart]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-apps.git
0f2d1a7..e3d16f6 HEAD -> security-scans
```

- 7.4. Return to the Jenkins tab in your browser, and click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build fails in the **Security Scan** stage.

The **Security Scan** stage fails because the container base image contains the CVE-2020-14583 vulnerability.

- ▶ 8. Update the base image of the application to solve the vulnerability issues.

- 8.1. Use the `oc import-image` command to import an updated image of OpenJDK.

```
[user@host shopping-cart]$ oc import-image ubi8/openjdk-11:1.3-10 \
--from=registry.access.redhat.com/ubi8/openjdk-11:1.3-10 --confirm
imagestream.image.openshift.io/openjdk-11 imported
...output omitted...
```

- 8.2. Use the `oc edit` command to edit the `BuildConfig`.

```
[user@host shopping-cart]$ oc edit bc/security-scans
```

The preceding command opens the `BuildConfig` in an editor. Update the OpenJDK image stream tag from `1.3-3` to `1.3-10` and save the changes.

```
...output omitted...
spec:
  ...output omitted...
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: openjdk-11:1.3-10
  ...output omitted...
```

- 8.3. Return to the Jenkins tab in your browser. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful.

- ▶ 9. Clean up.

- 9.1. Click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.
- 9.2. Return to the command line terminal, delete the OpenShift projects, return to the `main` branch, and navigate back to your workspace folder.

```
[user@host shopping-cart]$ oc delete project \
RHT_OCP4_DEV_USER-security-scans
project.project.openshift.io "youruser-security-scans" deleted
[user@host shopping-cart]$ git checkout main
Switched to branch 'main'
...output omitted...
[user@host deploying-envs]$ cd ../../
[user@host DO400]$
```

This concludes the guided exercise.

Defining Performance Metrics

Objectives

After completing this section, you should be able to identify and monitor pipeline performance metrics.

Measuring the Process

As you improve your processes and adopt more DevOps practices, you need ways to measure your progress. Capturing metrics around software delivery cannot tell you your agility or whether a particular practice is promoting long-term growth, but it can demonstrate *specific* improvements. For example, an improvement in the *mean time to recovery* metric might demonstrate an overall improvement in deployment practices.



Note

Do not overvalue a specific metric and accidentally turn it into a target. This *often* leads to a focus on solely improving that metric regardless of whether it drives business and user value.

Differentiating Outcomes and Outputs

As we discuss ways to gather metrics, be mindful of the difference between measuring outputs versus outcomes.

Your metrics and process improvements should aim to create positive outcomes. This will inherently mean an improvement in output, but perhaps not in the short term.

Outcomes

Outcomes refer to desired business and user changes, usually achieved by a change in process, methodology, or culture. Outcomes are more useful for long-term change. Examples of outcomes include the following:

- Improving user experience
- Reducing product defects
- Reducing time to market

Although outcomes are expressed like metrics, they are more qualitative than outputs. They are usually difficult to measure directly.

Outputs

Outputs are the tangible products of work or change. Examples of outputs include the following:

- Lines of code
- New features of an application

- Bugs fixed
- Number of story points completed

Do not be fooled into measuring outputs by themselves. Instead, focus on what drives positive outcomes.

Consider a product with a browser-based user interface and a desired outcome to reduce product defects. Although it is beneficial to track the number of bugs fixed over time, this is an output. It does not necessarily indicate an overall reduction in defects if a lot of those defects are not software related. You might need to find additional solutions to drive your outcome, such as user experience testing.

The Watermelon Effect

Although achieving an outcome often requires an improvement in output, **only increasing output does not necessarily lead to a desired outcome.**

This is known as the *watermelon effect*. The watermelon effect is when the status of a project seems to perform well when it is actually failing. The project seems green from the outside, but is red on the inside, like a watermelon.



Note

Similarly, be mindful of Goodhart's Law, which is summarized to be: "when a measure becomes a target, it ceases to be a good measure."

Describing Four Software Delivery Metrics

In the book *Accelerate: The Science of Lean Software and DevOps*, the authors measure and explore various metrics of highly successful software businesses and teams. In their studies, they observed four metrics surrounding software delivery performance.

These metrics are interconnected with each other. No one metric is mutable in isolation, therefore they must all improve together to have any effect.

Agility Metrics

- *Deployment frequency* measures how often an application is deployed *to production*. The authors of *Accelerate* found that successful teams tended to make more frequent production deployments.
- *Lead time for change* focuses on how long it takes for a code change to be deployed **in production**. Specifically, it measures the amount of time between a code change being committed and being available to users.

Reliability Metrics

- *Failure rate* is the percentage of failed or broken releases. The other metrics should not come at the cost of releasing broken software to customers.
- *Mean time to recovery* is the average amount of time required to recover from a failed release.

A potential outcome associated with the preceding agility metrics is a decrease in *batch size*, which is the volume of changes delivered in a given release and should be as small as possible.

For the reliability metrics, a potential outcome is an increase in stability.

Gathering Data

To track these metrics, it is best to automate their gathering, querying, and display.

Potential data sources include the following, among others:

- CI pipeline tooling, such as Jenkins
- Cloud platforms, such as OpenShift, AWS, and Kubernetes
- Git forges, such as GitHub and GitLab
- Project management tools, such as Jira and ZenHub

Metrics Tools

- **Prometheus** is an open source data aggregation and query tool. It can be configured to collect data from various sources, such as Jenkins. Once data is gathered, you can use Prometheus' query language, PromQL, to pull data and calculate metrics. You can also use these queries to set up alerts.
- **Grafana** is an open source graph and visualization framework. It provides functionality for building and displaying custom graphs and dashboards. It can be configured to display data directly from Prometheus.

Connecting Jenkins to Prometheus

You can configure Jenkins as a source by using the Jenkins Prometheus plug-in. Once installed, this plug-in exposes metrics at the `<JENKINS_URL>/prometheus` endpoint.

Connecting Jenkins as a source enables you to query pipeline metrics using PromQL. For example, the following query expression evaluates to the total number of milliseconds the `your-app` pipeline took on all of its runs.

```
default_jenkins_builds_duration_milliseconds_summary_sum{jenkins_job="your-app"}
```

Although none of the tooling provides the four metrics discussed earlier by default, you can use PromQL to construct a query for each of them.

For example, the following query calculates the `lead time for change` for the pipeline named `end-to-end-pipeline`, in seconds.

```
default_jenkins_builds_duration_milliseconds_summary_sum{jenkins_job="end-to-end-pipeline"} /
default_jenkins_builds_duration_milliseconds_summary_count{jenkins_job="end-to-end-pipeline"} /
1000
```

In the accompanying guided exercise, you will install and configure Prometheus and Grafana on OpenShift to measure the `lead time for change` and `mean time to recovery` for an example application.



Note

OpenShift comes with cluster monitoring via Prometheus preconfigured. However, to use it on the provided shared cluster, you must set up Prometheus separately. This is covered in the guided exercise.



References

Outcomes vs Outputs: What's the Difference?

<https://www.bmc.com/blogs/outcomes-vs-outputs/>

Using Metrics to Guide Container Adoption, Part I

<https://www.openshift.com/blog/using-metrics-to-guide-container-adoption-part-i>

Exploring a Metrics-Driven Approach to Transformation

<https://www.openshift.com/blog/exploring-a-metrics-driven-approach-to-transformation>

Accelerate: The Science of Lean Software and DevOps

<https://itrevolution.com/book/accelerate/>

Monitoring in OpenShift 4.6

<https://docs.openshift.com/container-platform/4.6/monitoring/understanding-the-monitoring-stack.html>

Jenkins Prometheus Plug-in

<https://plugins.jenkins.io/prometheus/>

► Guided Exercise

Defining Performance Metrics

In this exercise you will gather metrics to measure your software delivery performance.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (```) instead of the backslash character (`\`) to introduce line breaks in these commands.

Outcomes

You should be able to set up Grafana and Prometheus to gather DevOps metrics from Jenkins.

Before You Begin

To perform this exercise, ensure you have:

- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- The OpenShift CLI



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Prepare Jenkins to expose metrics to Prometheus.
 - 1.1. Open the Jenkins dashboard in a web browser.
 - 1.2. Click **Manage Jenkins** in the left pane, and then click **Configure Global Security**.
 - 1.3. Scroll down to the **Authorization** area. In the **Anonymous Users** table row, select the Read check box of the **Job** column. Click **Save**.

Authorization

Strategy

Authorization

☐ Anyone can do anything
 ☐ Legacy mode
 ☐ Logged-in users can do anything
 ☒ Matrix-based security

	Overall	Credentials	Agent	Job	Run	
User/group	Administer	ManageDomains Create Delete Read Update	View Configure Connect Create Delete Disconnect Build Cancel Configure	Discover Delete Create Move Read Workspace	Replay Delete Update Configure	
Anonymous Users	☐	☐ ☐ ☐ ☐ ☐	☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐	☐ ☐ ☐ ☐ ☒ ☐ ☐ ☐ ☐ ☐	☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐	

**Note**

Metrics are provided by the Prometheus Jenkins plug-in, which is preinstalled in your Jenkins instance.

Normally, you would set up a dedicated account for Prometheus to read metrics from Jenkins. The exercise uses anonymous access for the sake of simplicity.

► 2. Install Prometheus.

- 2.1. Open a command-line terminal, log in to OpenShift and create a new project for this exercise.

```
[user@host D0400]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
[user@host D0400]$ oc new-project RHT_OCP4_DEV_USER-metrics
Now using project "youruser-metrics"
on server "https://api.cluster.example.com:6443".
```

- 2.2. Get the host name of the Jenkins instance.

```
[user@host D0400]$ oc get route jenkins \
-o jsonpath="{.spec.host}" \
-n RHT_OCP4_DEV_USER-jenkins
jenkins-your-user-jenkins.apps.cluster.example.com
```

- 2.3. Deploy Prometheus to OpenShift by using the `D0400-apps/tools/prometheus-template.yml` file. This template sets up Prometheus to scrape metrics from a Jenkins instance every 15 seconds. Replace `YOUR_JENKINS_HOST` with the Jenkins host that you just retrieved.

```
[user@host D0400]$ oc process -f D0400-apps/tools/prometheus-template.yml \
-p JENKINS_HOST=YOUR_JENKINS_HOST \
| oc apply -f -
configmap/prometheus-config created
deployment.apps/prometheus created
service/prometheus created
route.route.openshift.io/prometheus created
```

The preceding command processes a template that includes the **Deployment**, **ConfigMap**, **Service** and **Route** resources. The **Deployment** specifies the desired state of the Prometheus application by using the `quay.io/prometheus/prometheus` image. The **ConfigMap** contains the Prometheus configuration, including the route to the Jenkins instance. The **Service** and the **Route** expose the application.

2.4. Get the Prometheus URL.

```
[user@host D0400]$ oc get route prometheus \
-o jsonpath='{\"http://\"}{.spec.host}{\"\\n\"}'
http://prometheus-your-user-metrics.apps.cluster.example.com
```

- 2.5. Open the Prometheus URL in a web browser tab to verify that the Prometheus application works. In the navigation bar, click **Status** → **Targets** and check that the `jenkins` endpoint is up.

▶ 3. Install Grafana and connect it with Prometheus.

- 3.1. Deploy and expose Grafana by using the `registry.redhat.io/openshift4/ose-grafana` image.

```
[user@host D0400]$ oc new-app registry.redhat.io/openshift4/ose-grafana
...output omitted...
--> Success
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/ose-grafana'
Run 'oc status' to view your app.
[user@host D0400]$ oc expose svc/ose-grafana
route.route.openshift.io/ose-grafana exposed
```

3.2. Get the Grafana URL.

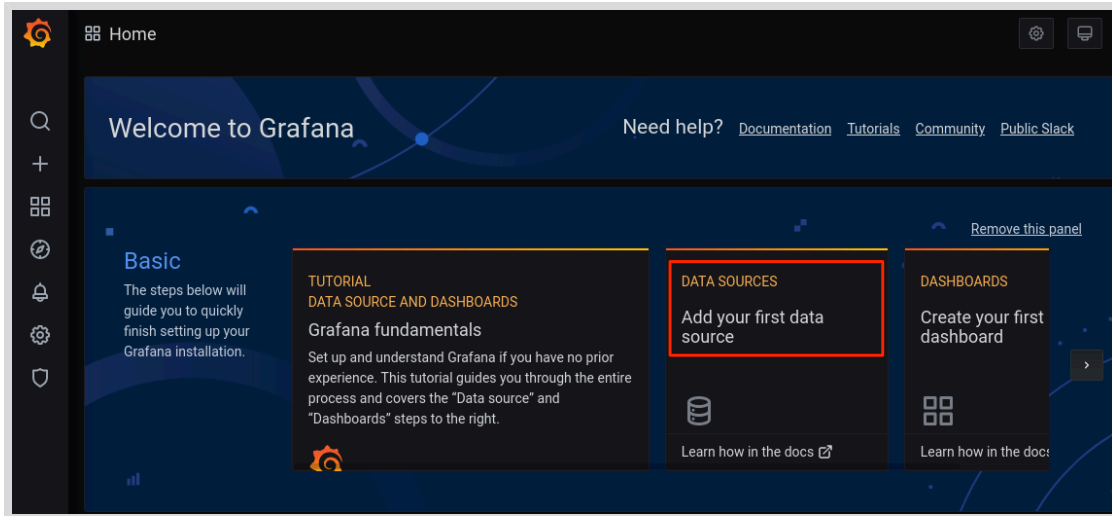
```
[user@host D0400]$ oc get route ose-grafana \
-o jsonpath='{\"http://\"}{.spec.host}{\"\\n\"}'
http://ose-grafana-your-user-metrics.apps.cluster.example.com
```

- 3.3. Open a new browser tab and navigate to the Grafana URL. Log in by using `admin` for the user name and `admin` for the password. Skip the step to change the password.

**Note**

The exercise skips the password change step for simplicity. Normally, you should create a specific admin account and remove the default one.

- 3.4. The Grafana home page shows some steps to quickly set up the application. Click the **Add your first data source** step.



- 3.5. From the **Time series databases** list, select **Prometheus**.
- 3.6. Under the **HTTP** section, enter `http://prometheus:9090` for the URL.
- 3.7. Click **Save and test**. A success message is displayed to confirm that the integration with Prometheus works.
- ▶ 4. Create a Jenkinsfile and deploy the sample application from the `D0400-apps/greeting-cd-pipeline` folder.
 - 4.1. Navigate to the `D0400-apps/greeting-cd-pipeline` folder and checkout the main branch of the `D0400-apps` repository to ensure you start from a known clean state.

```
[user@host D0400]$ cd D0400-apps/greeting-cd-pipeline
[user@host greeting-cd-pipeline]$ git checkout main
...output omitted...
Your branch is up to date with 'origin/main'.
[user@host simple-calculator]$ git pull origin main
...output omitted...
Already up to date.
```

- 4.2. Create a new branch to save any changes you make during this exercise.

```
[user@host greeting-cd-pipeline]$ git checkout -b metrics
Switched to a new branch 'metrics'
```

- 4.3. Open the Jenkinsfile and replace its contents with the following:

```

pipeline {
  agent { label 'nodejs' }

  stages{
    stage('Test'){
      steps {
        dir('greeting-cd-pipeline') {
          sh "node test.js"
        }
      }
    }

    stage('Deploy') {
      steps {
        sh '''
          oc start-build greeting-metrics \
            --follow --wait -n RHT_OCP4_DEV_USER-metrics
        '''
      }
    }
  }
}

```

Replace RHT_OCP4_DEV_USER with your lab user.

- 4.4. Save the file, commit, and push the changes to the branch.

```

[user@host greeting-cd-pipeline]$ git commit Jenkinsfile -m "add deployment"
[metrics fdf0b72] add deployment
1 file changed, 16 insertions(+), 13 deletions(-)
[user@host greeting-cd-pipeline]$ git push origin metrics
...output omitted...
1e64f78..fdf0b72  metrics -> metrics

```

- 4.5. Create a new application called `greeting-metrics`. Use the code in the `greeting-cd-pipeline` folder of the `metrics` branch from your repository.

```

[user@host greeting-cd-pipeline]$ oc new-app --name greeting-metrics \
https://github.com/YOUR_GITHUB_USER/DO400-apps#metrics \
--strategy=docker \
--context-dir=greeting-cd-pipeline
...output omitted...
--> Success
Build scheduled, use 'oc logs -f bc/greeting-metrics' to track its progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/greeting-metrics'
Run 'oc status' to view your app.

```

- 4.6. Expose the application.

```
[user@host greeting-cd-pipeline]$ oc expose svc/greeting-metrics
route.route.openshift.io/greeting-metrics exposed
```

4.7. Get the URL of the application.

```
[user@host greeting-cd-pipeline]$ oc get route greeting-metrics \
-o jsonpath="{ 'http://' }{ .spec.host }{ '\n' }"
http://greeting-metrics-your-user-metrics.apps.cluster.example.com
```

Open a new browser tab and navigate to the URL. Verify that the application returns the `Hello student` response. You might need to wait a few seconds until the application is completely deployed.

► **5.** Create a new pipeline in Jenkins for the `greeting-metrics` application.

5.1. Allow Jenkins to run deployments in your `RHT_OCP4_DEV_USER-metrics` project.

```
[user@host greeting-cd-pipeline]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins \
-n RHT_OCP4_DEV_USER-metrics
clusterrole.rbac.authorization.k8s.io/view added: "system:serviceaccount:your-
user-jenkins:jenkins"
```

5.2. Return to the Jenkins browser tab. Navigate to the Jenkins main page. Click **New Item**.

5.3. Enter `greeting-metrics` for the name, select **Pipeline** for the type, and click **OK**.

5.4. In the **Pipeline** section, select or enter the following:

Field	Value
Definition	Pipeline script from SCM
SCM	Git
Repository URL	<code>https://github.com/YOUR_GITHUB_USERNAME/DO400-apps.git</code>
Branch Specifier	<code>*/metrics</code>
Script Path	<code>greeting-cd-pipeline/Jenkinsfile</code>

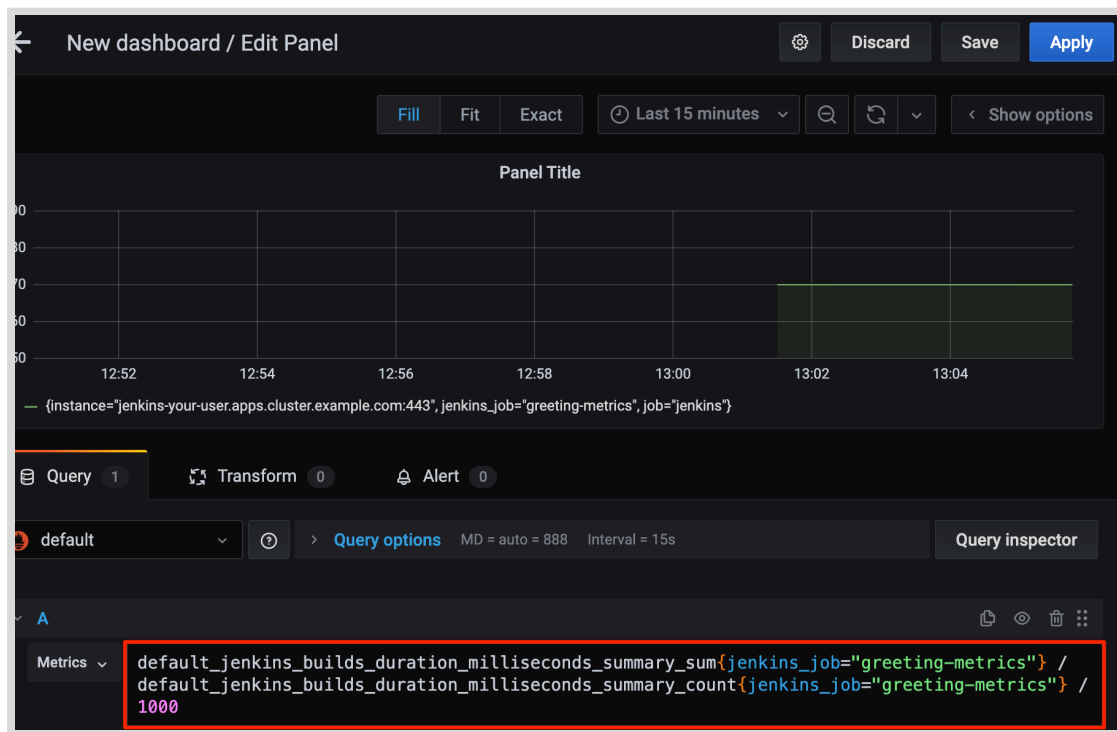
Click **Save** to save the changes.

5.5. From the menu on the left, click **Build Now**. The build should finish successfully.

► **6.** In Grafana, add a dashboard and a panel to show the average pipeline duration metric.

6.1. Return to the Grafana browser tab. In the left navigation pane, click **+**, then click **+ Add new panel**. The **Edit Panel** view opens.

- 6.2. Enter the PromQL query. In the **Query** tab, specify the following PromQL query in the input field next to the **Metrics** selector.



```
default_jenkins_builds_duration_milliseconds_summary_sum{jenkins_job="greeting-
metrics"} /
default_jenkins_builds_duration_milliseconds_summary_count{jenkins_job="greeting-
metrics"} /
1000
```

Because the average pipeline duration is not a metric provided by Jenkins directly, you must calculate it by using the following metrics:

- `default_jenkins_builds_duration_milliseconds_summary_sum` is the sum of the durations of all pipeline builds, in milliseconds.
- `default_jenkins_builds_duration_milliseconds_summary_count` is the number of executed pipeline builds.

Therefore, the average pipeline duration is the sum of the durations of all pipeline builds divided by the total number of pipeline builds. The query uses the `jenkins_job` parameter to gather metrics only for the `greeting-metrics` pipeline. Finally, the query divides the result by 1000, to get the metric in seconds.



Note

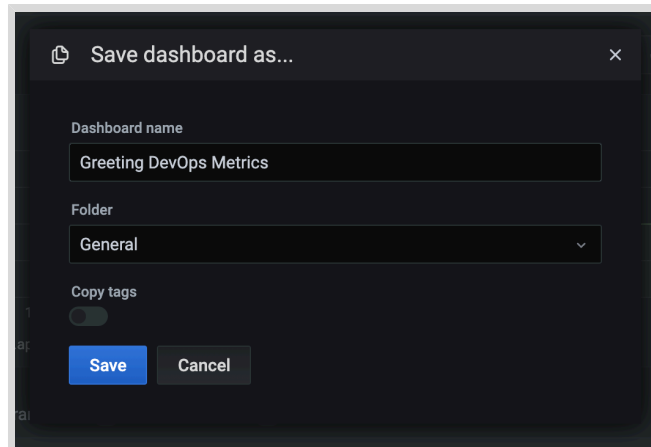
PromQL queries are interpreted and executed by Prometheus, which stores the metrics scraped from Jenkins. Grafana only acts as an interface.

- 6.3. Press **Shift + Enter** to show the metric graph. Observe the generated metric.

**Note**

The average pipeline duration is an agility metric, which is closely related to the **Lead Time for Change** metric. The Lead Time for Change shows how long it takes for committed changes to be reviewed, integrated, tested, and deployed to production.

- 6.4. In the right pane, enter **Average CI/CD Duration** for the panel title.
- 6.5. Click **Save** in the upper right. In the window that opens, enter **Greeting DevOps Metrics** for the dashboard name and click **Save**.



- ▶ 7. In Jenkins, create a new pipeline to roll back a deployment.
 - 7.1. Return to the Jenkins browser tab and navigate to the Jenkins main page. Click **New Item**.
 - 7.2. Enter **greeting-metrics-rollback** for the name, select **Pipeline** for the type, and click **OK**.
 - 7.3. Select the **This build is parameterized** check box. Add a new string parameter named **COMMIT**.
 - 7.4. Scroll down to the **Pipeline** section and specify the following pipeline in the **Script** field.

```
pipeline {
    agent any
    parameters {
        string(name: "COMMIT")
    }
    stages {
        stage('Rollback') {
            steps {
                script {
                    if (params.COMMIT == null) {
                        error("Please specify the COMMIT parameter")
                    }
                }
            }
        }
    }
    sh ""
```

```

        oc start-build greeting-metrics \
        --commit ${params.COMMIT} --follow --wait \
        -n RHT_OCP4_DEV_USER-metrics
    ""
    }
}
}
}

```

Replace `RHT_OCP4_DEV_USER` with your lab user. Click **Save** to save the changes. Do not run the pipeline yet. You will run it in a subsequent step, after a bug is introduced.

- ▶ **8.** In Grafana, add a new panel to measure the average duration of the Rollback pipeline.
 - 8.1. Return to your Grafana browser tab. Make sure that you are in **Greeting DevOps Metrics** dashboard by using the dashboard selector in the upper left. In the top actions bar, click the **Add panel** button. In the new pane that shows, click + **Add new panel**.
 - 8.2. In the input field next to the **Metrics** selector, specify the following PromQL query.

```

default_jenkins_builds_duration_milliseconds_summary_sum{jenkins_job="greeting-
metrics-rollback"} /
default_jenkins_builds_duration_milliseconds_summary_count{jenkins_job="greeting-
metrics-rollback"} /
1000

```

This metric shows the average `greeting-metrics-rollback` pipeline duration in seconds. In this case, this is equal to measuring how long it takes to roll back changes in production.



Note

The preceding metric is closely related with the **Mean Time to Recovery** metric.

In more complex scenarios, you will probably want to also measure the time to recover from a failure in production.

- 8.3. Press **Shift + Enter**. The graph shows no data because you have not yet executed your first rollback.
- 8.4. In the right pane, enter **Average Rollback Duration** for the panel title.
- 8.5. Click **Save** in the upper right. In the window that opens, click **Save** again. Click **Apply** to return to the dashboard.
- ▶ **9.** Introduce an error in the code and deploy the changes.
 - 9.1. Open the `greeting-cd-pipeline/server.js` file and modify the code to return a server error. Apply the following changes:

```

...output omitted...
const server = http.createServer((req, res) => {
  const { name } = url.parse(req.url, true).query;

```

```
res.statusCode = 500;
res.setHeader("Content-Type", "text/plain");
res.setHeader("Access-Control-Allow-Origin", "*");
res.end(greet("Error"));
});
...output omitted...
```

9.2. Save the file, commit, and push the changes to the branch.

```
[user@host greeting-cd-pipeline]$ git commit server.js -m "error"
[metrics 9ca7cd8] error
1 file changed, 2 insertions(+), 2 deletions(-)
[user@host greeting-cd-pipeline]$ git push origin metrics
...output omitted...
1e64f78..fdf0b72 metrics -> metrics
```

9.3. Return to the Jenkins browser tab. Navigate to the `greeting-metrics` pipeline details page. Click **Build Now** to deploy the changes. Wait for the pipeline to finish.

9.4. Get the URL of the application.

```
[user@host greeting-cd-pipeline]$ oc get route greeting-metrics \
-o jsonpath="{ 'http://' }{ .spec.host }{ '\n' }"
http://greeting-metrics-your-user-metrics.apps.cluster.example.com
```

If you already have the URL open in a browser tab, refresh the page. Otherwise, navigate to the application URL and verify that the application returns an error.

► 10. Run the `greeting-metrics-rollback` pipeline to roll back the changes.

10.1. Return to the Jenkins browser tab and navigate to the `greeting-metrics-rollback` pipeline details page. Click **Build with Parameters** to roll back the last deployment. Enter `HEAD~1` as the **COMMIT** parameter and click **Build**. This will roll back the deployment to the previous commit.



Note

In Git, the `HEAD` pointer refers to the last commit on the current branch.

To refer to previous commits, use `HEAD~`, followed by a number. For example, `HEAD~1` is a reference to the previous commit.

10.2. Wait for the pipeline build to finish. Verify that the application works again.

► 11. In Grafana, verify that the **Average Rollback duration** graph shows data. You might need to wait a few seconds until Prometheus gets the latest Jenkins metrics. Observe how long it takes to run the `greeting-metrics-rollback` pipeline.

► 12. Add a metric to measure the rate of pipelines per hour.

12.1. From the dashboard, click **Add panel** button.

12.2. Specify the following query.

```
rate(default_jenkins_builds_duration_milliseconds_summary_count{jenkins_job="greeting-metrics"}[1h])
```

The `rate` function returns the average value of a counter in a specific interval. Because Prometheus scrapes data from Jenkins every 15 seconds, the `default_jenkins_builds_duration_milliseconds_summary_count` is in fact a time series. The `rate` function averages the time series data points over the specified interval, which is one hour in this case.

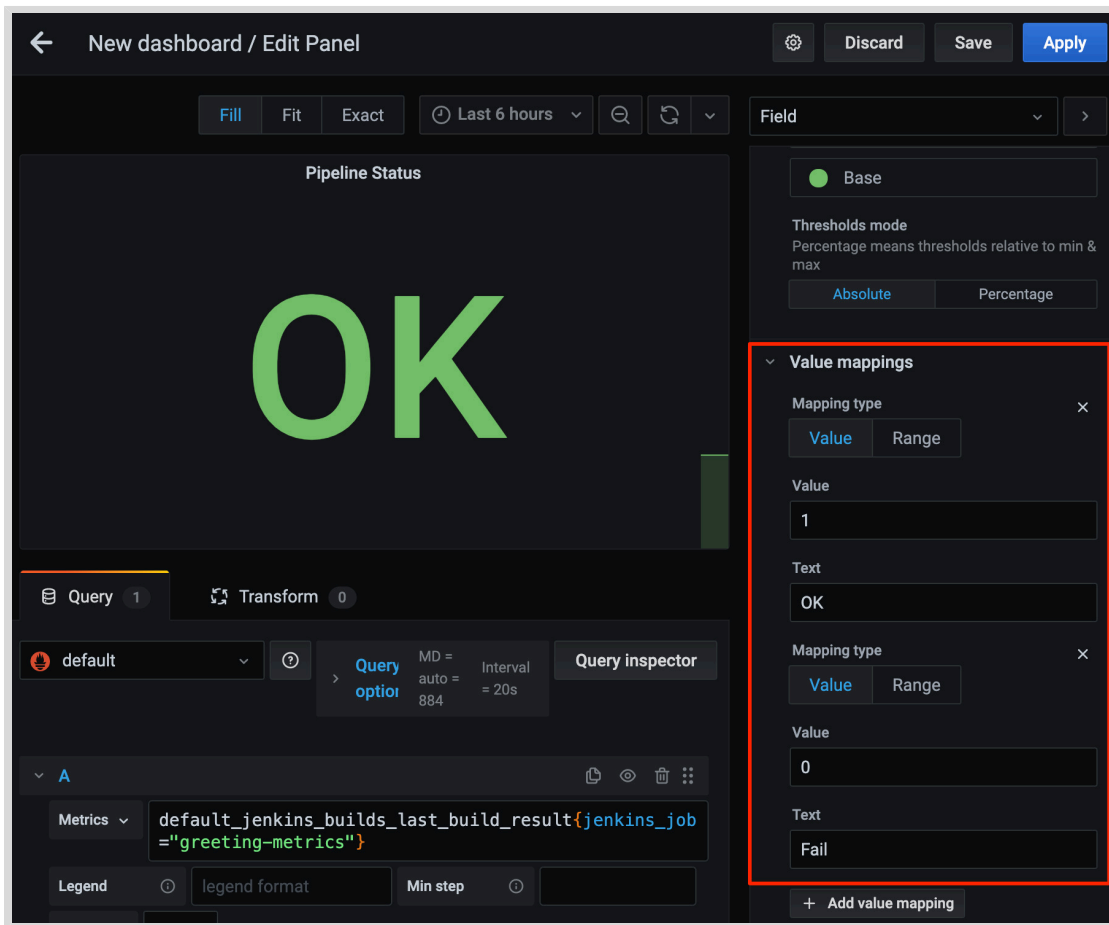
- 12.3. Press **Shift + Enter** to show the metric graph.
- 12.4. In the right pane, enter `CD pipeline builds/hour` for the panel title.
- 12.5. Click **Save** in the upper right. In the window that opens, click **Save** again. Click **Apply** to return to the dashboard.

► **13.** Add another metric to show the success status of the `greeting-metrics` pipeline.

- 13.1. Create another panel by using the following query:

```
default_jenkins_builds_last_build_result{jenkins_job="greeting-metrics"}
```

- 13.2. Press **Shift + Enter** to show the metric graph. You should see a time series graph showing a value of one.
- 13.3. In the right pane, enter `Pipeline Status` for the panel title.
- 13.4. In the right pane, select the `Stat` visualization type.
- 13.5. In the right pane, click **Field**. Scroll down to the **Value mappings** section. Click **Add value mapping**. Enter `1` for the value and `OK` for the text. Notice how the graph changes.
- 13.6. Add another value mapping. Enter `0` for the value and `Fail` for the text.



- 13.7. Click **Save** in the upper right. In the window that opens, click **Save** again. Click **Apply** to return to the dashboard.

Note how this indicator quickly shows the health status of the pipeline.

► **14.** Clean up.

- 14.1. Return to the Jenkins tab in your browser, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.
- 14.2. Return to the command-line terminal, and delete the OpenShift project.

```
[user@host greeting-cd-pipeline]$ oc delete project \
RHT_OCP4_DEV_USER-metrics
project.project.openshift.io "your-user-metrics" deleted
```

- 14.3. Checkout the **main** branch of the **D0400-apps** repository and return to your workspace folder.

```
[user@host greeting-cd-pipeline]$ git checkout main
[user@host greeting-cd-pipeline]$ cd ~/D0400
[user@host D0400]$
```

This concludes the guided exercise.

Configuring Error Notifications and Alerts

Objectives

After completing this section, you should be able to configure Jenkins alerts and error notifications.

Describing Alerts

An important part of the DevOps mindset is the continuous monitoring of your software life cycle. Monitoring your platform and workflow allows you to discover problems and react to them faster. Instead of continuously watching your platform, you can set up alerts to keep track of certain events.

Generally, you want to raise an alert for events that might negatively affect your workflow, system, and business performance. The following are examples of conditions under which you might want to raise an alert:

- The unit testing stage of a CI/CD pipeline fails.
- A node of your production platform is down.
- Your application is below the desired number of replicas in production.

Cloud platforms, such as Red Hat OpenShift, often include dedicated systems to manage alerts. Normally, you can create alerts based on thresholds, health checks, operating ranges, and severity levels, among others. For example, you might want to trigger a critical alert if your application is unresponsive for more than five seconds. Managing alerts for cloud platform monitoring is outside the scope of this course.

In CI/CD tools, alerts usually inform about undesired outcomes in a pipeline build. As well as triggering an alert when a pipeline fails, it is common to send an alert when a pipeline has been fixed. Some stages warrant an alert on every execution, such as a production deployment.

Notifying Alerts with Jenkins

To announce the alert, you must use a notification mechanism, which is a channel through which an alert or other kinds of messages are communicated. A simple example of a notification is an email.

You can generally decide your preferred notification channel to notify alerts. Most CI/CD tools allow you to use different notification mechanisms, such as emails and messages via third-party messaging applications. In Jenkins, this is done via plug-ins. The following are popular notification plug-ins:

- Mailer plug-in
- Slack plug-in
- Google Chat Notification plug-in

In addition to alerts, you can use notifications as a way to send informative messages about the execution status of a pipeline. You can also combine different plug-ins for different notification

types. For example, you can rely on chat plug-ins for instant alert and error notifications, and leave emails for less critical announcements, such as deployment notifications.

Configuring the Mailer Plug-in

You can use the Jenkins Mailer plug-in to send email notifications from your pipeline scripts. Before using the plug-in, you need an SMTP server with which to send emails. Next, you must specify the server address in the **E-mail Notification** section of the Jenkins **Configure System** page. The **Advanced** button shows additional authentication parameters.

E-mail Notification

SMTP server: your-smtp-server.example.com

Default user e-mail suffix:

☐ Use SMTP Authentication

☐ Use SSL

SMTP Port:

Reply-To Address:

Charset: UTF-8

☒ Test configuration by sending test e-mail

Test e-mail recipient: test@example.com

Test configuration

Figure 7.8: Jenkins email notification settings



Note

The Mailer plug-in is preinstalled in the Jenkins instance that you use in this course.

Sending Emails From Pipelines

After you configure the SMTP server, you are ready to start sending emails. To send an email from a pipeline, use the `mail` step as shown in the following example.

```
mail to: "pablo@example.com",
      subject: "Pipeline error",
      body: "The following error has occurred: ...output omitted..."
```

The `mail` step requires the `to`, `body`, and `subject` parameters. You can also use the `cc` and `bcc` parameters to send a carbon copy to other recipients. If you want to send the notification to multiple email addresses, separate the addresses with commas.

Notifying Pipeline Status

Similar to other pipeline steps, you can use the `mail` step in any stage of the pipeline. Usually, it is a good idea to use this step in the `post` section. The `post` section defines additional tasks to execute upon the completion of a pipeline, or a stage. For example, you can send an email alert in case of pipeline failure, as the following example shows:

```

post {
    failure {
        mail to: "pablo@example.com",
            subject: "Pipeline error",
            body: "The following error has occurred: ...output omitted..."
    }
}

```

This example uses the `failure` section to send an email when the pipeline fails. Likewise, you can use the `success` section to send an email notification when the pipeline succeeds. If you want to notify the team about a pipeline going back to a successful state, then use the `fixed` section.

You can also use the `mail` step within a pipeline script.

```

stage('Optional stage') {
    steps {
        script {
            try {
                sh './optional-script'
            } catch (Exception e) {
                mail to: "devs@example.com",
                    subject: "Warning",
                    body: "An optional stage failed"
            }
        }
    }
}

```

This example shows how you can use the `mail` step for noncritical alerts. In this case, if the `./optional-script` command throws an exception, then Jenkins sends a *warning* email.



References

Jenkins Mailer plug-in

<https://plugins.jenkins.io/mailer/>

Pipeline Basic Steps: Mail

<https://www.jenkins.io/doc/pipeline/steps/workflow-basic-steps/#mail-mail>

Pipeline Syntax: Post

<https://www.jenkins.io/doc/book/pipeline/syntax/#post>

Jenkins Google Chat Notification plug-in

<https://plugins.jenkins.io/google-chat-notification/>

Jenkins Slack Notification plug-in

<https://plugins.jenkins.io/slack/>

► Guided Exercise

Configuring Error Notifications and Alerts

In this exercise you will create a pipeline, send email notifications, and capture stage failures to perform specific actions.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to send email notifications and alerts in different steps of the pipeline.

Before You Begin

To perform this exercise, ensure you have:

- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- The OpenShift CLI



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the notifications of the `simple-calculator` application, and create an environment in OpenShift to deploy the application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps/simple-calculator` application folder. Checkout the `main` branch of your `D0400-apps` fork to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps/simple-calculator
[user@host simple-calculator]$ git checkout main
...output omitted...
Your branch is up to date with 'origin/main'.
[user@host simple-calculator]$ git pull origin main
...output omitted...
Already up to date.
```

- 1.2. Create a new branch named `notifications` to save any changes you make during this exercise. Push the branch to your GitHub fork.

```
[user@host simple-calculator]$ git checkout -b notifications
Switched to a new branch 'notifications'
[user@host simple-calculator]$ git push origin HEAD
...output omitted...
* [new branch] HEAD -> notifications
```

- 1.3. Log in to OpenShift with the credentials provided in the Lab Environment tab of the Red Hat online learning website. You can also copy the login command from the OpenShift web console.

```
[user@host simple-calculator]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
```

- 1.4. Create an OpenShift project to host the `simple-calculator` application.

```
[user@host simple-calculator]$ oc new-project RHT_OCP4_DEV_USER-notifications
Now using project "youruser-notifications" on server "https://
api.cluster.domain.example.com:6443".
...output omitted...
```

- 1.5. Add the `edit` role to the Jenkins account in the `RHT_OCP4_DEV_USER-notifications` project.

```
[user@host simple-calculator]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-
jenkins:jenkins"
```

Notice that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user.

- 1.6. Deploy in OpenShift an SMTP server that captures the email notification and alerts.

```
[user@host simple-calculator]$ oc apply -f kubefiles/email-service.yaml
deployment.apps/email created
service/email created
route.route.openshift.io/email created
```

- 1.7. Get the SMTP server route.

```
[user@host simple-calculator]$ oc get route email \
-o jsonpath="{ 'http://' }.spec.host"{'\n'}"
http://email-your-user-notifications.apps.cluster.example.com
```

- 1.8. Open a browser and navigate to the URL that you just retrieved. You will use this web interface to view all the emails sent to the SMTP server.

- ▶ 2. Create a declarative pipeline composed of different stages executed sequentially. The stages for the pipeline are the following:

- An explicit checkout of the `notifications` branch.
- An execution of the application tests.

- 2.1. Create a file named `Jenkinsfile` in the root of the application folder to store the pipeline. The file contents should look similar to this:

```
pipeline {
  agent { node { label 'maven' } }
  options { skipDefaultCheckout true }
  stages {
    stage('Checkout') {
      steps {
        git branch: 'notifications',
            url: 'https://github.com/YOUR_GITHUB_USER/DO400-apps'
      }
    }
    stage('Test') {
      steps {
        dir('simple-calculator') {
          sh './mvnw clean test'
        }
      }
    }
  }
}
```

Declarative pipelines automatically check out code from source control. The `skipDefaultCheckout` option disables that default action.

- 2.2. Commit and push the `Jenkinsfile` file.

```
[user@host simple-calculator]$ git add Jenkinsfile
[user@host simple-calculator]$ git commit -m "Added Jenkins integration"
[notifications 77b5cd9] Added Jenkins integration
1 file changed, 19 insertions(+)
create mode 100644 Jenkinsfile
[user@host simple-calculator]$ git push origin HEAD
...output omitted...
32ac625..77b5cd9 HEAD -> notifications
```

- ▶ 3. Configure Jenkins to send email notifications and alerts.

- 3.1. Open a new tab in the browser and navigate to the URL of the Jenkins server that you installed in *Guided Exercise: Configuring a Developer Environment*. If prompted, log in into Jenkins with your OpenShift credentials.
- 3.2. Click **Manage Jenkins** in the left pane, and then click **Configure System** in the right pane.
- 3.3. Scroll down to the **E-mail Notification** area to configure an SMTP server.
- 3.4. Type `email.RHT_OCP4_DEV_USER-notifications` in the **SMTP server** field.
- 3.5. Select the **Test configuration by sending test e-mail** check box, and then type `test@example.com` in the **Test e-mail recipient** field.
- 3.6. Click **Test configuration**. A success message displays to confirm that the testing email was sent.
- 3.7. Click **Save**.
- 3.8. Return to the SMTP web interface tab in your browser. Verify that the web interface displays a test email sent to `test@example.com` in the left pane.
- 3.9. Click **Clear Inbox** to remove all the stored emails.

► **4.** Create a Jenkins project to run the pipeline.

- 4.1. Return to the Jenkins tab in your browser.
- 4.2. In the left pane, click **New Item**.
- 4.3. Assign the name `do400-notifications` to the project, then click **Pipeline**, and finally click **OK**.
- 4.4. Select the **Discard old builds** check box, and then type 3 in the **Max # of builds to keep** field.
- 4.5. Click the **Pipeline** tab. In the **Pipeline** area, from the **Definition** list, select **Pipeline script from SCM**.
- 4.6. From the **SCM** list select **Git**. In the **Repository URL** type the URL of the GitHub repository of your `DO400-apps` fork.
The URL of the repository is similar to `https://github.com/YOUR_GITHUB_USER/DO400-apps`.
- 4.7. In the **Branch Specifier** field, type `*/notifications`.
- 4.8. In the **Script Path** field, type `simple-calculator/Jenkinsfile`.
- 4.9. Click **Save** to save the pipeline and go to the pipeline details page.
- 4.10. From the left side navigation, click **Build Now** to schedule a pipeline run. Wait for the build to finish and verify that it is successful.

► **5.** Send an email notification to `team@example.com` when the build starts.

- 5.1. Open the `Jenkinsfile` file and append a method to send email notifications to the end of the file. The code should look similar to this:

```

pipeline {
    ...output omitted...
    stages {
        ...output omitted...
    }
}

void sendNotification(String action, String email) {
    mail to: "${email}",
        subject: "Pipeline ${action}: ${currentBuild.fullDisplayName}",
        body: "The following pipeline ${action}: ${env.BUILD_URL}"
}

```

5.2. Add a call to the `sendNotification` method in the Checkout stage.

```

pipeline {
    ...output omitted...
    stages {
        stage('Checkout') {
            steps {
                sendNotification('started', 'team@example.com')
                git branch: 'notifications',
                    url: 'https://github.com/YOUR_GITHUB_USER/DO400-apps'
            }
        }
        ...output omitted...
    }
}
...output omitted...

```

5.3. Commit and push the changes.

```

[user@host simple-calculator]$ git commit -a -m "Added start notification"
[notifications a77e087] Added start notification
1 file changed, 7 insertions(+)
[user@host simple-calculator]$ git push origin HEAD
...output omitted...
77b5cd9..a77e087 HEAD -> notifications

```

5.4. Return to the Jenkins tab in your browser, and then click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful.

5.5. Open to the SMTP web interface tab in your browser. Verify that the web interface displays an email to `team@example.com` with the subject **Pipeline started**.

5.6. Click **Clear Inbox** to remove all the stored emails.

► 6. Send an email notification to `team@example.com` when the build finishes.

6.1. Open the `Jenkinsfile` file and add a `post` section after the `stages` section. The code should look similar to this:

```

pipeline {
    agent { node { label 'maven' } }
    options { skipDefaultCheckout true }
    stages {
        ...output omitted...
    }
    post {
        always {
            sendNotification('finished', 'team@example.com')
        }
    }
}

void sendNotification(String action, String email) {
    ...output omitted...
}

```

The `post` section defines additional steps, which run on the completion of a pipeline execution. The `always` condition runs the encapsulated actions regardless of the completion status of the pipeline execution.



Note

You can use the `post` section to define actions to execute after a stage or a pipeline.

6.2. Commit and push the changes.

```

[user@host simple-calculator]$ git commit -a -m "Added finish notification"
[notifications a77e087] Added finish notification
1 file changed, 5 insertions(+)
[user@host simple-calculator]$ git push origin HEAD
...output omitted...
77b5cd9..a77e087 HEAD -> notifications

```

6.3. Return to the Jenkins tab in your browser, and then click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful.

6.4. Open to the SMTP web interface tab in your browser. Verify that the web interface displays the following:

- An email to `team@example.com` with the subject `Pipeline started`.
- An email to `team@example.com` with the subject `Pipeline finished`.

6.5. Click **Clear Inbox** to remove all the stored emails.

► 7. Add a `Check Style` stage to the pipeline, and send an email alert to `team@example.com` when the pipeline execution fails.

7.1. Open the `Jenkinsfile` and append a method to send email alerts. The code should look similar to this:

```

pipeline {
    ...output omitted...
}

void sendNotification(String action, String email) {
    ...output omitted...
}

void sendFailureAlert(String email) {
    mail to: "${email}",
        subject: "Pipeline failed: ${currentBuild.fullDisplayName}",
        body: "The following pipeline failed: ${env.BUILD_URL}"
}

```

7.2. Add a Check Style stage after the Test stage. The code should look similar to this:

```

pipeline {
    ...output omitted...
    stages {
        ...output omitted...
        stage('Test') {
            ...output omitted...
        }
        stage('Check Style') {
            steps {
                dir('simple-calculator') {
                    sh './mvnw clean checkstyle:check'
                }
            }
        }
    }
    ...output omitted...
}

```

7.3. Add a failure condition in the post section.

```

pipeline {
    ...output omitted...
    post {
        failure {
            sendFailureAlert('team@example.com')
        }
        always {
            sendNotification('finished', 'team@example.com')
        }
    }
}
...output omitted...

```

The failure condition executes the encapsulated actions when the pipeline build fails.

7.4. Commit and push the changes.

```
[user@host simple-calculator]$ git commit -a -m "Added style stage"
[notifications 83f4a78] Added finish notification
1 file changed, 16 insertions(+)
[user@host simple-calculator]$ git push origin HEAD
...output omitted...
a827cc3..83f4a78 HEAD -> notifications
```

- 7.5. Return to the Jenkins tab in your browser, and then click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build fails in the **Check Style** stage.
- 7.6. Open to the SMTP web interface tab in your browser. Verify that the web interface displays the following:
 - An email to `team@example.com` with the subject `Pipeline started`.
 - An email to `team@example.com` with the subject `Pipeline finished`.
 - An email to `team@example.com` with the subject `Pipeline failed`.
- 7.7. Click **Clear Inbox** to remove all the stored emails.
- 8. Send a notification about the failure of the `Check Style` stage to `devs@example.com`.
 - 8.1. Open the `Jenkinsfile` and append a method to send warning alerts. The code should look similar to this:

```
...output omitted...

void sendFailureAlert(String email) {
    ...output omitted...
}

void sendWarningAlert(String stage, String email) {
    mail to: "${email}",
    subject: "${stage} stage failed: ${currentBuild.fullDisplayName}",
    body: "The ${stage} stage failed: ${env.BUILD_URL}"
}
```

- 8.2. Update the `Check Style` stage to capture any exception thrown by the `checkstyle:check` Maven action, send a warning alert to `devs@example.com`, and set the build status as `Unstable`.

```
...output omitted...
stage('Check Style') {
    steps {
        dir('simple-calculator') {
            script {
                try {
                    sh './mvnw clean checkstyle:check'
                } catch (Exception e) {
                    sendWarningAlert("${STAGE_NAME}", 'devs@example.com')
                    unstable("${STAGE_NAME} stage failed!")
                }
            }
        }
    }
}
```



```

    }
  }
}
...output omitted...

```

8.3. Commit and push the changes.

```

[user@host simple-calculator]$ git commit -a -m "Added style warning"
[notifications 5151282] Added style warning
1 file changed, 14 insertions(+), 1 deletion(-)
[user@host simple-calculator]$ git push origin HEAD
...output omitted...
223dcba..5151282 HEAD -> notifications

```

8.4. Return to the Jenkins tab in your browser, and then click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build result is **Unstable**.

8.5. Open to the SMTP web interface tab in your browser. Verify that the web interface displays the following:

- An email to `team@example.com` with the subject `Pipeline started`.
- An email to `team@example.com` with the subject `Pipeline finished`.
- An email to `devs@example.com` with the subject `Check Style stage failed`.

8.6. Click **Clear Inbox** to remove all the stored emails.

► 9. Fix the code standards issues.

9.1. Open the `Jenkinsfile` file and change the branch used in the pipeline to `solutions`. The code should look similar to this:

```

pipeline {
  ...output omitted...
  stages {
    stage('Checkout') {
      steps {
        sendNotification('started', 'team@domain.com')
        git branch: 'solutions',
            url: 'https://github.com/YOUR_GITHUB_USER/D0400-apps'
      }
    }
  }
  ...output omitted...
}

```

9.2. Commit and push the changes.

```
[user@host simple-calculator]$ git commit -a -m "Fixed style errors"
[notifications ad5839d] Fixed style errors
1 file changed, 1 insertion(+), 1 deletion(-)
[user@host simple-calculator]$ git push origin HEAD
...output omitted...
61ff357..ad5839d HEAD -> notifications
```

- 9.3. Return to the Jenkins tab in your browser, and then click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build is successful. Refresh the browser to verify that the latest artifacts are available in Jenkins.
- 9.4. Open to the SMTP web interface tab in your browser. Verify that the web interface displays the following:
 - An email to `team@example.com` with the subject `Pipeline started`.
 - An email to `team@example.com` with the subject `Pipeline finished`.

► **10.** Clean up.

- 10.1. Return to the Jenkins tab in your browser, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.
- 10.2. Return to the command-line terminal. Delete the OpenShift project, switch back to the `main` branch, and navigate back to your workspace folder.

```
[user@host simple-calculator]$ oc delete project \
RHT_OCP4_DEV_USER-notifications
project.project.openshift.io "your-user-notifications" deleted
[user@host simple-calculator]$ git checkout main
Switched to branch 'main'
[user@host simple-calculator]$ cd ../../
[user@host D0400]$
```

This concludes the guided exercise.

Recovering Failures

Objectives

After completing this section, you should be able to decrease failure rates and rollback failed deployments.

Automating Ways to Recover from Failures

As you automate more of your process, you should also automate ways to recover from unexpected bugs, breakages, and failures. Ideally, tests would catch potential errors, but there are always unforeseen scenarios. Eventually, you *will* release a severe bug to production.

Before you can mitigate a failure, you must first detect it. There are a number of tools and techniques to discover flaws before and after they are deployed to production. These error detection techniques are either *proactive* or *reactive*.

Proactive techniques occur *before* an issue is actually observed in production, and *reactive techniques* occur *after*. One guideline to tell the difference is whether a user *could have* experienced the bug before it was detected.

Proactive techniques in particular are an example of the DevOps mindset of **failing fast**. The earlier you detect it, the cheaper and easier it is to correct. Also, it is always better to detect an issue before users experience it.

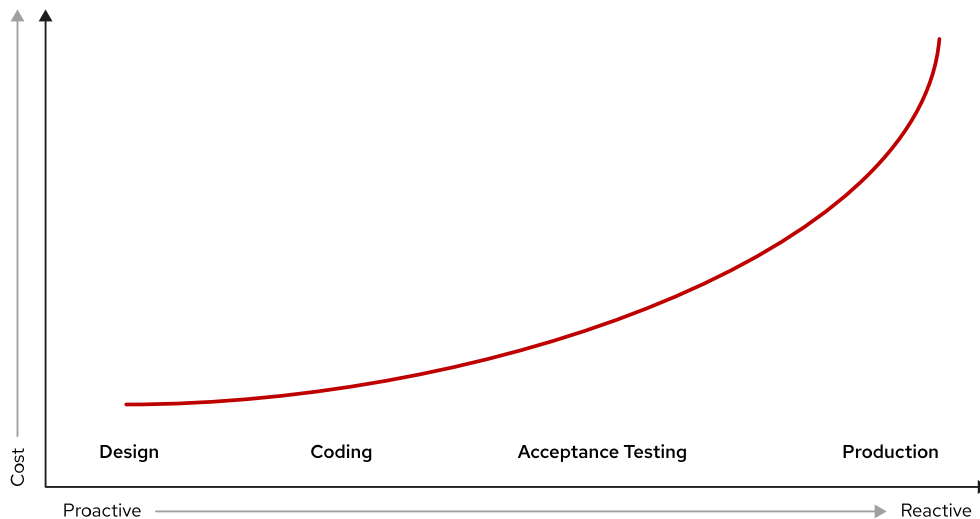


Figure 7.9: The cost of fixes over time

Retries

Perhaps the simplest proactive technique is to *retry* the operation. For example, if a deployment fails, it could be due to any number of confounding variables, such as network congestion. In these cases, the failed operation just needs another attempt.

Keep in mind that this should not be your *only* solution. If it is a reoccurring pattern, you must attempt to diagnose and correct the underlying issue.

To add a retry using a Jenkins pipeline, use `retry`. For example, the following snippet tries to run the test script up to three times before failing the step.

```
retry(3) {
    sh './mvnw clean test'
}
```

Timeouts

A *timeout* occurs any time an operation fails for taking more time than a configured threshold. Timeouts are usually proactive because they help to detect unoptimized or misconfigured operations and processes.

Pipeline stages are a common place to use timeouts. For example, if a test stage takes too long, then that might be a sign that they do not have proper mocks or that further optimization is needed. When used in a pipeline, timeouts are proactive because they trigger before the affected code is deployed to production.

As part of the accompanying guided exercise, you will configure a Jenkins pipeline that includes such a timeout. Many testing frameworks default to a sensible timeout value. Depending on which framework you use, you might need to configure your pipeline to detect such errors.

In your Jenkins pipeline, you can add a timeout with `timeout`. For example, the following snippet runs the test script with a timeout of ten minutes.

```
timeout(time: 10, unit: 'MINUTES') {
    sh './mvnw clean test'
}
```



Note

The `timeout` function is available at multiple levels within your pipeline, including within a `steps` block. Additionally, it can be added to an `options` block to add a timeout for the entire stage.

Security Scans

Security scans are another proactive technique for detecting potential issues before they occur.

A security scanning tool looks for common vulnerabilities in your application and infrastructure. For example, Quay.io includes container image vulnerability scanning. Your pipelines should incorporate at least one security scanning suite.

For more details around security scans and how they relate to **DevSecOps**, reference *Implementing the Basic Principles of DevSecOps*.

Smoke Tests

As you have seen in other sections, automated testing goes a long way to assuring the validity of your code. Once you have an integration or functional test suite, you can run those tests on newly

deployed environments. This method of validating deployments is often called a *smoke test* and is handy for tracking down troublesome code changes and deployment errors.

**Note**

The term `smoke test` comes from a couple of places:

- Testing mechanical systems by filling them with visible smoke in order to check for leaks
- Testing electronics for the first time and looking for fire or smoke

You can run smoke tests on any or all of your non-production environments. These smoke tests are proactive, because they run on code before it is released to production. As you get closer to production, consider running more tests.

Depending on circumstances, you might use any or all types of testing as smoke tests. For example, you might only run unit tests in your **DEV** environment and *all* of your different test suites on your **PRE-PRODUCTION** environment.

Typically, you do not run automated tests on your production environment, because it uses real user data and resources. However, it is acceptable to run a carefully selected set of tests on dummy data for the purposes of smoke testing. These would be reactive because users could have experienced the issue before detection.

Deployment Strategies

Deployment strategies can help you discover issues with code changes before they are released to production. Specifically, `blue-green`, `shadow`, and `canary` deployments all act as a form of smoke test. These deployment strategies aim to both highlight defects, as well as minimize their impact.

For more detail around different deployment strategies, see *Implementing Release Strategies*.

Alerting and Monitoring

As you saw in *Configuring Error Notifications and Alerts*, alerting and monitoring helps to detect errors after they occur, which means they are always reactive.

For example, poorly optimized code might use higher resources than expected, which could trigger a predefined alert. Perhaps there is a deployment issue, so your pipeline should send a notification.

Another technique is to incorporate a monitoring tool that reports errors from a user's browser. Once configured, these tools automatically collect JavaScript errors. You can use the data collected to triage and fix errors without the need for user feedback.

Do not solely rely on your users to report bugs nor their ability to properly document the circumstances. After all, *you* are the one with access to the systems.

Rollbacks

Although your goal should be to minimize the number of errors that make it to production, they are an inevitability. When this happens, there are a number of ways to minimize their impact. For example, your chosen deployment strategy might reduce or remove traffic to affected environments.

Additionally, you can perform a manual *rollback*, which is where you deploy a previous version of an application in the event of an error. Exactly how you initiate a rollback depends on your deployment process. One of the simplest methods is to include functionality within your deployment pipeline that allows you to specify a version to deploy. In that case, if you need to perform a rollback, just enter an older version to be deployed.

You will build and use an example of this type of manual rollback in the accompanying guided exercise.

Adding Test Cases

Once you have detected and fixed an error, you should add a test case. This test should fail under the conditions before the fix and succeed once the mitigating changes are applied.

Adding a specific test case not only makes your test suite more robust, it assures you will not experience exactly the same bug again. In the case of a regression, your new test will now proactively detect the issue.



References

DevSecOps: Image scanning in your pipelines using quay.io scanner

<https://www.redhat.com/sysadmin/using-quayio-scanner>

Jenkins retry documentation

<https://www.jenkins.io/doc/pipeline/steps/workflow-basic-steps/#retry-retry-the-body-up-to-n-times>

Jenkins: Running multiple steps

<https://www.jenkins.io/doc/pipeline/tour/running-multiple-steps/>

Mechanical smoke tests

[https://en.wikipedia.org/wiki/Smoke_testing_\(mechanical\)](https://en.wikipedia.org/wiki/Smoke_testing_(mechanical))

Electrical smoke tests

[https://en.wikipedia.org/wiki/Smoke_testing_\(electrical\)](https://en.wikipedia.org/wiki/Smoke_testing_(electrical))

► Guided Exercise

Recovering Failures

In this exercise you will create a pipeline that prematurely stops the pipeline execution and allows you to recover quickly from failures.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (```) instead of the backlash character (`\`) to introduce line breaks in these commands.

Outcomes

You should be able to set timeouts to stages, stop pipeline execution early, and recover from failures.

Before You Begin

- A GitHub account
- Your D0400-apps fork cloned in your workspace
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- A web browser
- The OpenShift CLI
- Maven and a JDK
- A Quay.io account



Important

Try to keep your course work organized by creating a workspace folder, such as `~/D0400`. Start the guided exercise from this folder.

Instructions

- 1. Prepare your workspace to work on the recovering of the `shopping-cart-v2` application, and create an environment in OpenShift to deploy the application.
 - 1.1. From your workspace folder, navigate to the `D0400-apps/shopping-cart-v2` application folder. Checkout the `main` branch of your `D0400-apps` fork to ensure you start this exercise from a known clean state.

```
[user@host D0400]$ cd D0400-apps/shopping-cart-v2
[user@host shopping-cart-v2]$ git checkout main
...output omitted...
Your branch is up to date with 'origin/main'.
[user@host shopping-cart-v2]$ git pull origin main
...output omitted...
Already up to date.
```

- 1.2. Create a new branch named `recover` to save any changes you make during this exercise. Push the branch to your GitHub fork.

```
[user@host shopping-cart-v2]$ git checkout -b recover
Switched to a new branch 'recover'
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
* [new branch] HEAD -> recover
```

- 1.3. Log in to OpenShift with the credentials provided in the `Lab Environment` tab of the Red Hat online learning website. You can also copy the login command from the OpenShift web console.

```
[user@host shopping-cart-v2]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
```

- 1.4. Create an OpenShift project to host the `shopping-cart-v2` application.

```
[user@host shopping-cart-v2]$ oc new-project RHT_OCP4_DEV_USER-recover
Now using project "youruser-recover" on server "https://
api.cluster.domain.example.com:6443".
...output omitted...
```

- 1.5. Deploy the `shopping-cart-v2` application manually.

```
[user@host shopping-cart-v2]$ oc process \
-n RHT_OCP4_DEV_USER-recover \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
| oc apply -n RHT_OCP4_DEV_USER-recover -f -
deployment.apps/shopping-cart-v2 created
service/shopping-cart-v2 created
route.route.openshift.io/shopping-cart-v2 created
```

- 1.6. Add the `edit` role to the Jenkins account in the `RHT_OCP4_DEV_USER-recover` project.


```
[user@host shopping-cart-v2]$ oc policy add-role-to-user \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-
jenkins:jenkins"
```

- ▶ 2. Create a declarative pipeline composed of different stages executed sequentially. The stages for the pipeline are the following:
 - An implicit checkout of the `recover` branch.
 - An execution of the application tests.
- 2.1. Create a file named `Jenkinsfile` in the root of the application folder to store the pipeline. The file contents should look similar to this:

```
pipeline {
  agent { node { label 'maven' } }
  environment { APP_NAMESPACE = 'RHT_OCP4_DEV_USER-recover' }
  stages {
    stage('Test') {
      steps {
        dir('shopping-cart-v2') {
          sh './mvnw clean test'
        }
      }
    }
  }
}
```

- 2.2. Commit and push the `Jenkinsfile` file.

```
[user@host shopping-cart-v2]$ git add Jenkinsfile
[user@host shopping-cart-v2]$ git commit -m "Added Jenkins integration"
[recover d132ca5] Added Jenkins integration
1 file changed, 13 insertions(+)
create mode 100644 shopping-cart-v2/Jenkinsfile
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
ce03bf0..d132ca5 HEAD -> recover
```

- ▶ 3. Create a Jenkins project to run the pipeline.
 - 3.1. Open a new tab in the browser and navigate to the URL of the Jenkins server that you installed in *Guided Exercise: Configuring a Developer Environment*. If prompted, log into Jenkins with your OpenShift credentials.
 - 3.2. In the left pane, click **New Item**.
 - 3.3. Assign the name `do400-recover` to the project, then click **Pipeline**, and finally click **OK**.
 - 3.4. Select the **Discard old builds** check box, and then type **3** in the **Max # of builds to keep** field.

- 3.5. Click the **Pipeline** tab. In the **Pipeline** area, from the **Definition** list, select **Pipeline script** from **SCM**.
- 3.6. From the **SCM** list select **Git**. In the **Repository URL** type the URL of the GitHub repository of your **D0400-apps** fork.
The URL of the repository is similar to `https://github.com/YOUR_GITHUB_USER/D0400-apps`.
- 3.7. In the **Branch Specifier** field, type `*/recover`.
- 3.8. In the **Script Path** field, type `shopping-cart-v2/Jenkinsfile`.
- 3.9. Click **Save** to save the pipeline and go to the pipeline details page.
- 3.10. From the left side navigation, click **Build Now** to schedule a pipeline run. Wait for the build to finish and verify that it is successful.
Notice the amount of time it takes for the **Test** stage to complete. The **Test** stage is slow because the application uses a slow storage.

► **4.** Add a time-out to the **Test** stage.

- 4.1. Open the **Jenkinsfile** file and add a time-out of 50 seconds to the **Test** stage.
The file contents should look similar to this:

```
...output omitted...
stage('Test') {
    options { timeout(time: 50, unit: 'SECONDS') }
    steps {
        dir('shopping-cart-v2') {
            sh './mvnw clean test'
        }
    }
}
...output omitted...
```

- 4.2. Commit and push the changes.

```
[user@host shopping-cart-v2]$ git commit -a -m "Added time-out"
[recover 345d29d] Added time-out
1 file changed, 1 insertion(+)
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
d132ca5..345d29d HEAD -> recover
```

- 4.3. Return to the **Jenkins** tab in your browser. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and check that the build failed.
- 4.4. Navigate to the build details page and then click **Console Output** in the left pane to see the output from each stage. Examine the build output. Notice that the execution abortion of the build is because of the time-out rule in the **Test** stage.

```
...output omitted...
[Pipeline] End of Pipeline
Timeout has been exceeded
Finished: ABORTED
```

- 4.5. The application code uses a slow implementation for the catalog storage. Decouple the tests from the catalog storage implementation to make the tests run faster.
- Open the `src/test/java/com/redhat/shopping/integration/blackbox/ShoppingCartTest.java` file and mock the catalog storage. The code should look like the following:

```
...output omitted...
@QuarkusTest
public class ShoppingCartTest {

    @BeforeAll
    public static void setup() {
        CatalogStorage mockStorage = Mockito.mock(InMemoryCatalogStorage.class);

        Mockito.when(mockStorage.containsKey(1)).thenReturn(true);
        Mockito.when(mockStorage.containsKey(2)).thenReturn(true);
        Mockito.when(mockStorage.containsKey(9999)).thenReturn(false);

        Mockito.when(mockStorage.get(1)).thenReturn(new Product(1, 100));
        Mockito.when(mockStorage.get(2)).thenReturn(new Product(2, 200));

        QuarkusMock.installMockForType(mockStorage, CatalogStorage.class);
    }

    ...output omitted...
```

Open the `src/test/java/com/redhat/shopping/integration/whitebox/ShoppingCartTest.java` file and mock the catalog storage. The code should look like the following:

```
...output omitted...
@Inject
CartService cartService;

@BeforeAll
public static void setup() {
    CatalogStorage mockStorage = Mockito.mock(InMemoryCatalogStorage.class);

    Mockito.when(mockStorage.containsKey(1)).thenReturn(true);
    Mockito.when(mockStorage.containsKey(9999)).thenReturn(false);

    Mockito.when(mockStorage.get(1)).thenReturn(new Product(1, 100));

    QuarkusMock.installMockForType(mockStorage, CatalogStorage.class);
}

...output omitted...
```

4.6. Commit and push the changes.

```
[user@host shopping-cart-v2]$ git commit -a -m "Mocking storage"
[recover 107ed35] Mocking storage
 2 files changed, 26 insertions(+)
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
345d29d..107ed35 HEAD -> recover
```

4.7. Return to the Jenkins tab in your browser, and navigate to the pipeline details page. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and verify that it is successful.► 5. Add a stage to the pipeline named **Build Image** to generate a container image and push it to Quay.

- 5.1. Click the **Jenkins** logo in the upper left to navigate to the Jenkins homepage.
- 5.2. Click **Manage Jenkins** in the left pane, and then click **Manage Credentials** in the right pane.
- 5.3. In the **Stores scoped to Jenkins** section of the right pane, click **Jenkins**.
- 5.4. Click **Global credentials** in the right pane. Check the existence of a credential named **QUAY_USER**. If the **QUAY_USER** credential exists, skip the next step.
- 5.5. Click **Add Credentials** in the left pane. From the **Kind** list, select **Username with password**. In the **Username** field, type your Quay user name. In the **Password** field, type your Quay password. In the **ID** field, type **QUAY_USER**, and then click **OK**.
- 5.6. Open the **Jenkinsfile** file and add a stage named **Build Image** to execute after the **Test** stage. The stage generates a container image and pushes it to Quay by using the credentials stored in Jenkins.

Examine the `scripts/include-container-extensions.sh` and `scripts/build-and-push-image.sh` files. Those files obfuscate the complexity of building and pushing images to Quay. You can use those scripts in the pipeline.

The content of the stage should look like the following:

```
...output omitted...
stage('Build Image') {
    environment { QUAY = credentials('QUAY_USER') }
    steps {
        dir('shopping-cart-v2') {
            sh './scripts/include-container-extensions.sh'
            sh '''
                ./scripts/build-and-push-image.sh \
                -u $QUAY_USR \
                -p $QUAY_PSW \
                -b $BUILD_NUMBER
            '''
        }
    }
}
...output omitted...
```

5.7. Commit and push the changes.

```
[user@host shopping-cart-v2]$ git commit -a -m "Added Image stage"
[recover c5e7cbf] Added Image stage
1 file changed, 14 insertions(+)
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
107ed35..c5e7cbf HEAD -> recover
```

- 5.8. Return to the Jenkins tab in your browser, and navigate to the pipeline details page. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and verify that it is successful.
- 5.9. Open a new tab in your web browser, and verify that a new repository named **do400-recover** exists in your Quay account. Navigate to the image repository page located at https://quay.io/repository/YOUR_QUAY_USER/do400-recover.
- 5.10. Click **Tags** in the left navigation pane. Verify that the repository contains at least two tags:
 - One tag named **latest**.
 - One tag prefixed with **BUILD**, and a number that matches the build number of the Jenkins pipeline execution.
- 5.11. Click **Settings** in the left navigation pane. Scroll down the **Repository Visibility** area, click **Make Public** and confirm.

► 6. Add a stage to the pipeline named **Deploy**, to deploy new versions of the application to OpenShift.

- 6.1. Examine the `scripts/tag-exists-in-quay.sh` file. The `tag-exists-in-quay.sh` script uses `curl` to verify the existence of a tag in a Quay repository.
- 6.2. Examine the `scripts/redeploy.sh` file. The `redeploy.sh` script adds a tag to the Deployment's Pod template. Only changes to the Deployment's Pod template trigger a Deployment rollout.
- 6.3. Open the `Jenkinsfile` file and add a stage named **Deploy** to execute after the **Build Image** stage.
 Use the `tag-exists-in-quay.sh` script to check if the `latest` image tag exists in Quay. If the image is not available in Quay, then finalize the pipeline execution.
 Use the `redeploy.sh` script to deploy new versions of the application.
 The file content should look like the following:

```
...output omitted...
stage('Deploy') {
  environment { QUAY = credentials('QUAY_USER') }
  steps {
    dir('shopping-cart-v2') {
      script {
        def status = sh(
          script: './scripts/tag-exists-in-quay.sh $QUAY_USR/do400-
recover latest",
          returnStatus: true
```

```

    )

    if (status != 0) {
        error("Tag not found in Quay!")
    }
}

sh './scripts/redeploy.sh $APP_NAMESPACE'
}
}
}
...output omitted...

```

6.4. Commit and push the changes.

```

[user@host shopping-cart-v2]$ git commit -a -m "Added Deploy stage"
[recover b31c37c] Added Deploy stage
1 file changed, 19 insertions(+)
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
c5e7cbf..b31c37c HEAD -> recover

```

6.5. Return to the Jenkins tab in your browser, and navigate to the pipeline details page. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and verify that it is successful.

6.6. Verify that the application was deployed again. Use the `oc` command to verify the age of the application pods.

```

[user@host shopping-cart-v2]$ oc get pods -n RHT_OCP4_DEV_USER-recover

```

NAME	READY	STATUS	RESTARTS	AGE
shopping-cart-v2-5bfc7b7df4-xgl5h	1/1	Running	0	1m06s

► 7. Add a parameter to the pipeline to deploy a specific container image.

7.1. Open the `Jenkinsfile` file and add a parameter named `DEPLOY_TAG`. The code should look like the following:

```

pipeline {
    agent { node { label 'maven' } }
    environment { APP_NAMESPACE = 'RHT_OCP4_DEV_USER-recover' }
    parameters {
        string(name: 'DEPLOY_TAG', description: 'Deploy to Tag')
    }
    stages {
    ...output omitted...

```

7.2. Add a `when` expression to execute the `Test` and `Build Image` stages when the pipeline parameter is empty. The code should look like the following:

```

...output omitted...
stages {
    stage('Test') {

```

```

    when { expression { params.DEPLOY_TAG == '' } }
    ...output omitted...
  }
  stage('Build Image') {
    when { expression { params.DEPLOY_TAG == '' } }
    ...output omitted...
  }
  stage('Deploy') {
    ...output omitted...
  }
  ...output omitted...

```

- 7.3. Examine the `scripts/deploy-image.sh` file. The `deploy-image.sh` updates the image of the application container in the Deployment resource. This change triggers a Deployment rollout.
- 7.4. Update the `Deploy` stage to deploy a specific container image, or the latest build image generated in the pipeline. Remove the use of the `redploy.sh` script. Use the `deploy-image.sh` script to deploy new versions of the application specifying a tag. The code should look like the following:

```

...output omitted...
stage('Deploy') {
  environment { QUAY = credentials('QUAY_USER') }
  steps {
    dir('shopping-cart-v2') {
      script {
        def TAG_TO_DEPLOY = "BUILD-$BUILD_NUMBER"
        if (params.DEPLOY_TAG != '' && params.DEPLOY_TAG != null) {
          def status = sh(
            script: "./scripts/tag-exists-in-quay.sh $QUAY_USR/do400-
recover ${params.DEPLOY_TAG}",
            returnStatus: true
          )

          if (status != 0) {
            error("Tag not found in Quay!")
          }

          TAG_TO_DEPLOY = params.DEPLOY_TAG
        }

        sh """
          ./scripts/deploy-image.sh \
          -u $QUAY_USR \
          -n $APP_NAMESPACE \
          -t $TAG_TO_DEPLOY
        """
      }
    }
  }
}
...output omitted...

```

7.5. Commit and push the changes.

```
[user@host shopping-cart-v2]$ git commit -a -m "Optional stages"
[recover a1ed56a] Optional stages
1 file changed, 23 insertions(+), 8 deletions(-)
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
b31c37c..a1ed56a HEAD -> recover
```

7.6. Return to the Jenkins tab in your browser. In the left pane, click **Build Now** to schedule a pipeline run. Wait for the build to finish and reload the page.

7.7. In the left pane, click **Build with Parameters** to schedule a pipeline run. In the **DEPLOY_TAG** field type **404-NOT_FOUND**, and click **Build**.

Wait for the build to finish and verify the following:

- The build skipped the Test and Build Image stages.
- The build aborted the Deploy stage because the specified tag is not available in your Quay repository.

► 8. Add a new field to the response of the application.

8.1. Open the `src/main/java/com/redhat/shopping/cart/CartView.java` file. Add a new field named `snapshot` that contains the timestamp of the creation of the `CartView` object. The code should look like the following:

```
package com.redhat.shopping.cart;

import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Collection;

public class CartView {

    public ArrayList<CartItem> products;
    public int totalItems;
    public Timestamp snapshot;

    public CartView() {
    }

    public CartView(Collection<CartItem> products, int totalItems) {
        this.products = new ArrayList<>(products);
        this.totalItems = totalItems;
        this.snapshot = new Timestamp(System.currentTimeMillis());
    }
}
```

8.2. Run the tests and verify that the tests pass.


```
[user@host shopping-cart-v2]$ ./mvnw clean test
...output omitted...
[INFO]
[INFO] Tests run: 18, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

8.3. Commit and push the changes.

```
[user@host shopping-cart-v2]$ git commit -a -m "Added snapshot"
[recover 4a02ef2] Added snapshot
1 file changed, 3 insertions(+)
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
555aa1b..60c5318 HEAD -> recover
```

8.4. Return to the Jenkins tab in your browser. In the left pane, click **Build with Parameters** to schedule a pipeline run. Leave the **DEPLOY_TAG** field empty, and click **Build**. Wait for the build to finish and verify that it is successful.

8.5. Get the application route.

```
[user@host shopping-cart-v2]$ oc get route shopping-cart-v2 \
-o jsonpath="{ 'http://' + { .spec.host } + '/cart\n' }"
http://shopping-cart-v2-your-user-recover.apps.cluster.example.com/cart
```

8.6. Open a new browser tab and navigate to the URL that you just retrieved. Notice that the Quarkus application returns a JSON response composed of the `products` key with an empty list as value, the `totalItems` key with 0 as value, and the `snapshot` field.

► 9. Quickly undo the latest changes made to the deployed application.

9.1. Return to the Quay tab in your browser. Click **Tags** and reload the page. Choose an image tag that is not the latest one, and copy the tag name.

9.2. Return to the Jenkins tab in your browser. In the left pane, click **Build with Parameters** to schedule a pipeline run. In the **DEPLOY_TAG** field type the tag name selected in the preceding step, and click **Build**.

Wait for the build to finish and verify the following:

- The build skipped the **Test** and **Build Image** stages.
- The build executed the **Deploy** stage.

9.3. Return to the command-line terminal. Use the `oc get` command to examine the version deployed to OpenShift.

```
[user@host shopping-cart-v2]$ oc get deploy/shopping-cart-v2 -o \
jsonpath="{ .spec.template.spec.containers[0].image }{'\n'}"
quay.io/YOUR_QUAY_USER/do400-recover:BUILD-4
```

You quickly recovered to a previous stable version of the application. Now you can safely make corrections to your code without worrying about the production environment.

- 9.4. Open the `src/main/java/com/redhat/shopping/cart/CartView.java` file and undo the changes. The file should look like the following:

```
package com.redhat.shopping.cart;

import java.util.ArrayList;
import java.util.Collection;

public class CartView {

    public ArrayList<CartItem> products;
    public int totalItems;

    public CartView() {
    }

    public CartView(Collection<CartItem> products, int totalItems) {
        this.products = new ArrayList<>(products);
        this.totalItems = totalItems;
    }
}
```

Alternatively you can use the `git revert` command to restore the file to a previous state.

- 9.5. Commit and push the changes.

```
[user@host shopping-cart-v2]$ git commit -a -m "Removed snapshot"
[recover 791bbd2] Removed snapshot
1 file changed, 3 deletions(-)
[user@host shopping-cart-v2]$ git push origin HEAD
...output omitted...
555aa1b..60c5318 HEAD -> recover
```

- 9.6. Return to the Jenkins tab in your browser. In the left pane, click **Build with Parameters** to schedule a pipeline run. Leave the **DEPLOY_TAG** field empty, and click **Build**. Wait for the build to finish and verify that it is successful.
- 9.7. Return to the command-line terminal. Use the `oc get` command to examine the version deployed to OpenShift.

```
[user@host shopping-cart-v2]$ oc get deploy/shopping-cart-v2 -o \
jsonpath="{.spec.template.spec.containers[].image}"{'\n'}"
quay.io/YOUR_QUAY_USER/do400-recover:BUILD-10
```

Notice that the version deployed to OpenShift matches the build number in Jenkins.

► 10. Clean up.

- 10.1. Delete the OpenShift project, switch back to the `main` branch, and navigate back to your workspace folder.

```
[user@host shopping-cart-v2]$ oc delete project \
RHT_OCP4_DEV_USER-recover
project.project.openshift.io "your-user-recover" deleted
[user@host shopping-cart-v2]$ git checkout main
Switched to branch 'main'
[user@host shopping-cart-v2]$ cd ../../
[user@host DO400]$
```

- 10.2. Return to the Jenkins tab in your browser, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.

This concludes the guided exercise.

► Lab

Implementing Pipeline Security and Monitoring

In this lab, you will create a pipeline for a calculator application. In the pipeline you will use a private repository, send notification alerts on failures, scan container images for security vulnerabilities, and monitor the performance of the pipeline itself.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\\) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Access private Git repositories
- Send notification alerts
- Scan containers for security vulnerabilities
- Gather metrics from the pipeline executions

Before You Begin

To perform this exercise, ensure you have:

- Your D0400-apps fork cloned in your workspace folder
- A web browser such as Firefox or Chrome
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- Git installed
- A web browser
- The OpenShift CLI
- A Quay.io account



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start the guided exercise from this folder.

The calculator application used in this lab exposes a single endpoint `/solver/<equation>` that tries to solve the provided equation. This application can only solve sums and subtractions of decimal numbers. For example, the endpoint `/solver/4.02+8.0-1.1` returns the string `10.92`.

The source code for the application is available in the `calculator-monolith` folder in the GitHub repository at <https://github.com/RedHatTraining/DO400-apps>.

Make sure you start this lab from your workspace folder. Checkout the `main` branch of the `D0400-apps` fork to start this lab from a known clean state.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
...output omitted...
[user@host D0400-apps]$ cd ..
[user@host D0400]$
```

Instructions

1. Prepare the environment for the lab exercise.
Create a Red Hat OpenShift project named *RHT_OCP4_DEV_USER-monitoring-lab*, and grant `edit` permissions to the `jenkins` service account into the newly created project.
Use the `D0400-apps/tools/email-service.yml` template file to deploy an SMTP server that captures all email notifications and alerts. This SMTP server has a web interface you can use to verify the sending of emails.
2. Gather metrics from Jenkins to measure your software delivery performance.
Use the `D0400-apps/tools/prometheus-template.yml` file to deploy a Prometheus instance. Deploy and expose a Grafana instance by using the `registry.redhat.io/openshift4/ose-grafana` container image.
Connect Grafana to the Prometheus instance, and create a dashboard in Grafana named **Monitoring Lab Metrics**. Add a metric named **Average Pipeline Duration** to this dashboard. This metric displays the average pipeline duration metric by using the following PromQL query:

```
default_jenkins_builds_duration_milliseconds_summary_sum{
  jenkins_job="monitoring-lab"
} /
default_jenkins_builds_duration_milliseconds_summary_count{
  jenkins_job="monitoring-lab"
} / 1000
```

3. Store the application code in a private GitHub repository named `do400-monitoring-lab`. Use a folder named `monitoring-lab` as the root folder for this lab and the repository.
4. Create a public, empty Quay.io container image repository named `do400-monitoring-lab`.



Note

If you do not have an account in Quay.io, then create a new account to store container images for applications that you will build in this course. Refer to the detailed steps in the appendix *Creating a Quay Account*.

5. Store your GitHub and Quay.io credentials in Jenkins.
Create, and store a GitHub token to grant access to the private repository from the Jenkins pipeline. Name the GitHub token as `D0400_MONITORING_LAB` and grant the `repo` permissions. Store the token in Jenkins in a credential with global scope, assign `MONITORING_LAB_GH_CREDENTIALS` to the credential ID, and assign `MONITORING_LAB_GH` to the credential description.

Store your Quay.io credentials in Jenkins. Use a credential with global scope, assign `MONITORING_LAB_QUAY_CREDENTIALS` to the credential ID, and assign `MONITORING_LAB_QUAY` to the credential description.

6. Create a Jenkins project of the type **Pipeline** named `monitoring-lab`, and link it to the private GitHub repository. Use `*/main` as the branch specifier to only track changes in the main branch.
7. Create a pipeline that uses the declarative syntax, and follows the pipeline as code technique. The calculator is a Quarkus application that requires the maven agent to execute the pipeline.

The pipeline must include the following stages to execute in sequential order:

- **Test**: executes the `./mvnw clean test` command to validate the application.
- **Build**: uses the `./scripts/include-container-extensions.sh` script to add the required dependencies, and the `./scripts/build-and-push-image.sh` script to build and push a container image with the application to Quay.io.

The `build-and-push-image.sh` script requires the following parameters:

- `-b`: the build number
- `-u`: your Quay.io user
- `-p`: your Quay.io password
- `-r`: the Quay.io repository

8. Configure Jenkins to send email notifications and alerts by using the SMTP server deployed in a preceding step. Capture pipeline failures, and send email alerts to `team@example.com`. Append the pipeline name to the email subject, and embed the build URL into the email body.
9. Deploy the calculator application manually by using the `kubefiles/application-template.yml` template. Add a stage named **Deploy** that deploys the latest container image available in Red Hat OpenShift. Use the `scripts/redeploy.sh` script for deployments executed in the pipeline.

The `redeploy.sh` script requires the following parameters:

- `-d`: deployment name
- `-n`: namespace to use

10. Add security stage to scan the calculator container image for vulnerabilities. Name this stage **Security Scan**, and add it to the pipeline between the **Build** and **Deploy** stages.

Use the `kubefiles/security-scan-template.yml` template in combination with the `oc replace --force` command to create the Job that scans the container image.

Examine the template to find out the required parameters. Use the `scripts/check-job-state.sh` script to check the status of the scan job.



Note

Security experts find new vulnerabilities every day. This exercise only looks for one of them, `CVE-2021-23840`, to ensure that the result of the exercise remains deterministic.

In a real world scenario, you should check for all the vulnerabilities.

11. Clean up. Delete the Jenkins project, and the OpenShift project.

This concludes the lab.

► Solution

Implementing Pipeline Security and Monitoring

In this lab, you will create a pipeline for a calculator application. In the pipeline you will use a private repository, send notification alerts on failures, scan container images for security vulnerabilities, and monitor the performance of the pipeline itself.



Important

This exercise includes multi-line commands. Windows users must use the backtick character (``) instead of the backslash character (\) to introduce line breaks in these commands.

Outcomes

You should be able to:

- Access private Git repositories
- Send notification alerts
- Scan containers for security vulnerabilities
- Gather metrics from the pipeline executions

Before You Begin

To perform this exercise, ensure you have:

- Your D0400-apps fork cloned in your workspace folder
- A web browser such as Firefox or Chrome
- Access to a Jenkins server
- Access to the OpenShift shared cluster
- Git installed
- A web browser
- The OpenShift CLI
- A Quay.io account



Important

Try to keep your course work organized by creating a workspace folder, such as ~/D0400. Start the guided exercise from this folder.

The calculator application used in this lab exposes a single endpoint `/solver/<equation>` that tries to solve the provided equation. This application can only solve sums and subtractions of decimal numbers. For example, the endpoint `/solver/4.02+8.0-1.1` returns the string `10.92`.

The source code for the application is available in the `calculator-monolith` folder in the GitHub repository at <https://github.com/RedHatTraining/DO400-apps>.

Make sure you start this lab from your workspace folder. Checkout the `main` branch of the D0400-apps fork to start this lab from a known clean state.

```
[user@host D0400]$ cd D0400-apps
[user@host D0400-apps]$ git checkout main
...output omitted...
[user@host D0400-apps]$ cd ..
[user@host D0400]$
```

Instructions

1. Prepare the environment for the lab exercise.

Create a Red Hat OpenShift project named `RHT_OCP4_DEV_USER-monitoring-lab`, and grant edit permissions to the `jenkins` service account into the newly created project.

Use the `D0400-apps/tools/email-service.yml` template file to deploy an SMTP server that captures all email notifications and alerts. This SMTP server has a web interface you can use to verify the sending of emails.

- 1.1. Open a command-line terminal, and log in to Red Hat OpenShift by using the credentials provided in the `Lab Environment` tab of the course's online learning website.

```
[user@host D0400]$ oc login -u RHT_OCP4_DEV_USER \
-p RHT_OCP4_DEV_PASSWORD RHT_OCP4_MASTER_API
Login successful.
...output omitted...
```

- 1.2. Create a project named `RHT_OCP4_DEV_USER-monitoring-lab`.

```
[user@host D0400]$ oc new-project RHT_OCP4_DEV_USER-monitoring-lab
Now using project "youruser-monitoring-lab" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 1.3. Add the `edit` role to the `jenkins` service that is running in the `RHT_OCP4_DEV_USER-jenkins` project.

```
[user@host D0400]$ oc policy add-role-to-user \
-n RHT_OCP4_DEV_USER-monitoring-lab \
edit system:serviceaccount:RHT_OCP4_DEV_USER-jenkins:jenkins
clusterrole.rbac.authorization.k8s.io/edit added: "system:serviceaccount:youruser-jenkins:jenkins"
```

- 1.4. Deploy an SMTP server that captures the email notifications and alerts.

```
[user@host D0400]$ oc apply -f D0400-apps/tools/email-service.yml \
-n RHT_OCP4_DEV_USER-monitoring-lab
deployment.apps/email created
service/email created
route.route.openshift.io/email created
```

2. Gather metrics from Jenkins to measure your software delivery performance.

Use the `D0400-apps/tools/prometheus-template.yml` file to deploy a Prometheus instance. Deploy and expose a Grafana instance by using the `registry.redhat.io/openshift4/ose-grafana` container image.

Connect Grafana to the Prometheus instance, and create a dashboard in Grafana named **Monitoring Lab Metrics**. Add a metric named **Average Pipeline Duration** to this dashboard. This metric displays the average pipeline duration metric by using the following PromQL query:

```
default_jenkins_builds_duration_milliseconds_summary_sum{
  jenkins_job="monitoring-lab"
} /
default_jenkins_builds_duration_milliseconds_summary_count{
  jenkins_job="monitoring-lab"
} / 1000
```

- 2.1. Open a browser and navigate to the URL of the Jenkins server. If prompted, log in into Jenkins with your Red Hat OpenShift credentials.
- 2.2. Click **Manage Jenkins** in the left pane, and then click **Configure Global Security**.
- 2.3. Scroll down to the **Authorization** area. Make sure that, in the **Anonymous Users** table row, the **Read** check box of the **Job** column is selected, and click **Save**.
- 2.4. Return to the command-line terminal, and get the host name of the Jenkins instance.

```
[user@host D0400]$ oc get route jenkins \
-o jsonpath="{.spec.host}" \
-n RHT_OCP4_DEV_USER-jenkins
jenkins-your-user-jenkins.apps.cluster.example.com
```

Deploy a Prometheus instance by using the `D0400-apps/tools/prometheus-template.yml` file. Replace `YOUR_JENKINS_HOST` with the Jenkins host that you just retrieved.

```
[user@host D0400]$ oc process -f D0400-apps/tools/prometheus-template.yml \
-n RHT_OCP4_DEV_USER-monitoring-lab \
-p JENKINS_HOST=YOUR_JENKINS_HOST \
| oc apply -n RHT_OCP4_DEV_USER-monitoring-lab -f -
configmap/prometheus-config created
deployment.apps/prometheus created
service/prometheus created
route.route.openshift.io/prometheus created
```

- 2.5. Get the Prometheus URL.

```
[user@host D0400]$ oc get route prometheus \
-n RHT_OCP4_DEV_USER-monitoring-lab \
-o jsonpath="{ 'http://' }{.spec.host}" \
http://prometheus-your-user-metrics.apps.cluster.example.com
```

Open the Prometheus URL in a web browser tab to verify that the Prometheus application works. In the navigation bar, click **Status** → **Targets** and verify that the jenkins endpoint is up.

- 2.6. Return to the command-line terminal. Deploy and expose Grafana by using the `registry.redhat.io/openshift4/ose-grafana` image.

```
[user@host D0400]$ oc new-app registry.redhat.io/openshift4/ose-grafana \
-n RHT_OCP4_DEV_USER-monitoring-lab
...output omitted...
--> Success

Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/ose-grafana'

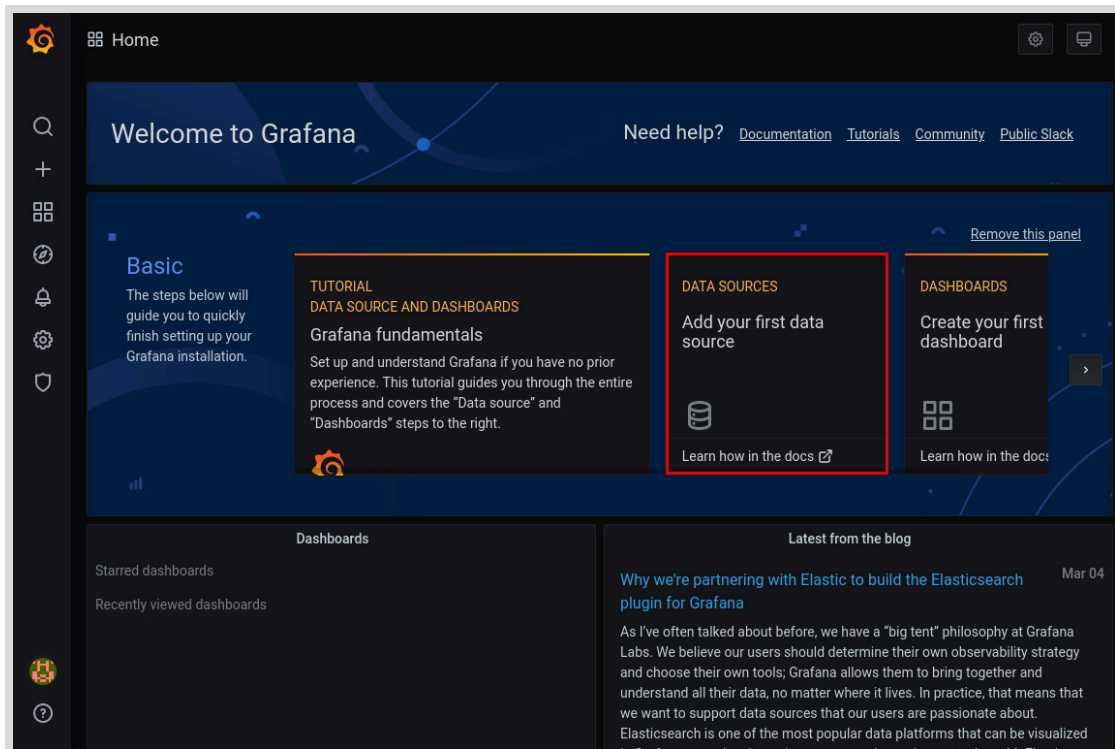
Run 'oc status' to view your app.
[user@host D0400]$ oc expose svc/ose-grafana
route.route.openshift.io/ose-grafana exposed
```

- 2.7. Get the Grafana URL.

```
[user@host D0400]$ oc get route ose-grafana \
-n RHT_OCP4_DEV_USER-monitoring-lab \
-o jsonpath="{ 'http://' }{ .spec.host }{ '\n' }"
http://ose-grafana-your-user-metrics.apps.cluster.example.com
```

Open a new browser tab and navigate to the Grafana URL. Log in by using `admin` for the user name and `admin` for the password. Click **Skip** to avoid changing the default password.

- 2.8. Click the **Add your first data source** step.



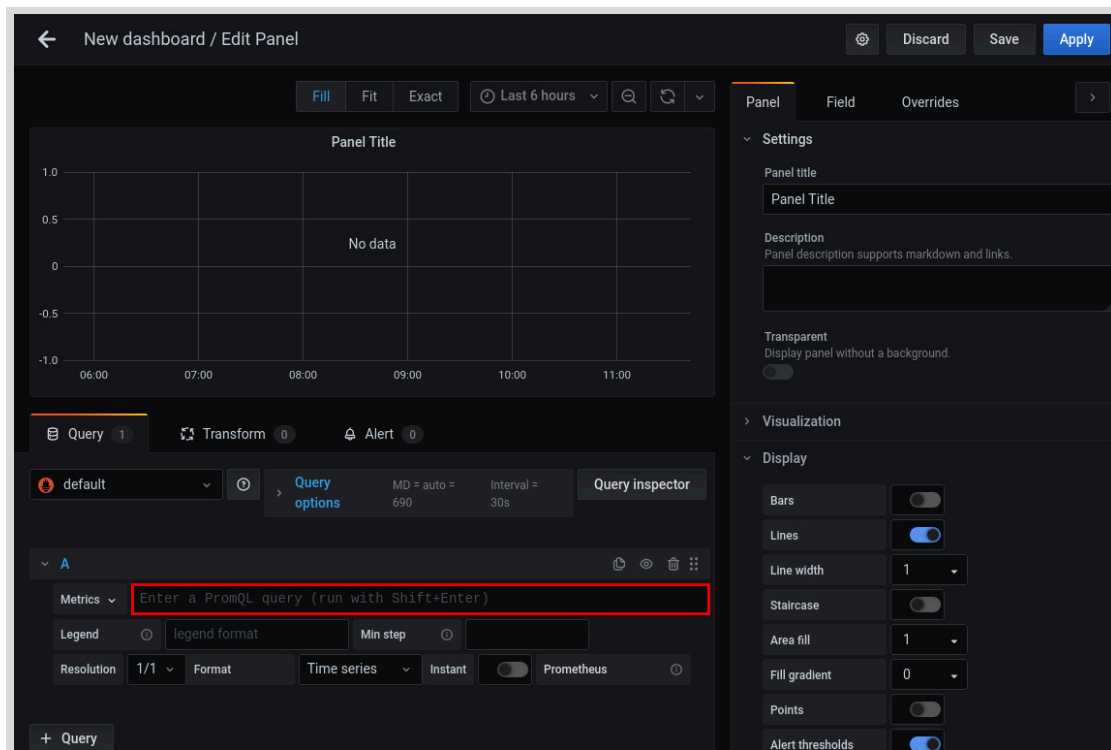
- 2.9. From the **Time series databases** list, select **Prometheus**.
- 2.10. In the **HTTP** area, type `http://prometheus:9090` in the URL field. Click **Save & Test**.

The success message `Data source is working` displays to confirm that the integration with Prometheus works.

- 2.11. In the left navigation pane, click **+**, then click **+ Add new panel**. The **Edit Panel** view opens.

In the **Query** tab, specify the following PromQL query in the input field next to the **Metrics** selector.

```
default_jenkins_builds_duration_milliseconds_summary_sum{
  jenkins_job="monitoring-lab"
} /
default_jenkins_builds_duration_milliseconds_summary_count{
  jenkins_job="monitoring-lab"
} / 1000
```



- 2.12. In the right pane, type **Average Pipeline Duration** in the **Panel title** field. Click **Save** in the upper right. In the window that opens, type **Monitoring Lab Metrics** in the **Dashboard name** field, and then click **Save**.



Note

Grafana will render the metrics generated by Jenkins while you perform this lab exercise.

3. Store the application code in a private GitHub repository named `do400-monitoring-lab`. Use a folder named `monitoring-lab` as the root folder for this lab and the repository.

- 3.1. Open a new tab in the browser, and navigate to `https://github.com`. If prompted, sign in with your GitHub credentials.

- 3.2. Click + → **New repository** in the upper right of the window. Then type `do400-monitoring-lab` in the **Repository name** field, select **Private**, and click **Create repository**.

**Note**

In the case where you are part of an organization, make sure that the repository is created under your GitHub user.

- 3.3. Return to the command-line terminal, and clone the private repository to your workspace. Clone the repository into a folder named `monitoring-lab`. If prompted, provide your GitHub credentials to clone the private repository.

```
[user@host D0400]$ git clone \
https://github.com/YOUR_GITHUB_USER/do400-monitoring-lab \
monitoring-lab
Cloning into 'monitoring-lab'
...output omitted...
warning: You appear to have cloned an empty repository.
```

- 3.4. Copy the application code located in `D0400-apps/calculator-monolith` to the repository folder you just created, and navigate to the `monitoring-lab` folder.

```
[user@host D0400]$ cp -Rv D0400-apps/calculator-monolith/. \
monitoring-lab
...output omitted...
[user@host D0400]$ cd monitoring-lab
[user@host monitoring-lab]$
```

**Important**

Windows users must replace the preceding `cp` command in PowerShell as follows:

```
PS C:\Users\user\D0400> Copy-Item -Path D0400-apps\calculator-monolith\* `
>> -Destination ~\D0400\monitoring-lab -Recurse
```

- 3.5. Commit and push the application code to the `main` branch of the remote repository.

```
[user@host monitoring-lab]$ git add -A
```

**Important**

Windows users must also execute the following commands after the `git add` command:

```
PS C:\Users\user\DO400\monitoring-lab> git update-index --chmod=+x mvnw
PS C:\Users\user\DO400\monitoring-lab> git update-index --chmod=+x `
>> scripts/include-container-extensions.sh
PS C:\Users\user\DO400\monitoring-lab> git update-index --chmod=+x `
>> scripts/build-and-push-image.sh
PS C:\Users\user\DO400\monitoring-lab> git update-index --chmod=+x `
>> scripts/redeploy.sh
PS C:\Users\user\DO400\monitoring-lab> git update-index --chmod=+x `
>> scripts/check-job-state.sh
```

```
[user@host monitoring-lab]$ git commit -m "Initial commit"
...output omitted...
[user@host monitoring-lab]$ git branch -M main
[user@host monitoring-lab]$ git push origin HEAD
...output omitted...
To https://github.com/your_github_user/do400-monitoring-lab.git
* [new branch]      HEAD -> main
```

4. Create a public, empty Quay.io container image repository named `do400-monitoring-lab`.

**Note**

If you do not have an account in Quay.io, then create a new account to store container images for applications that you will build in this course. Refer to the detailed steps in the appendix *Creating a Quay Account*.

- 4.1. Open a new tab in the browser, and navigate to `https://quay.io`. If prompted, sign in with your Quay.io credentials.
- 4.2. Click **Create New Repository** in the upper-right corner of the window.
- 4.3. Select **Container Image Repository** in the repository type field. In the **Repository name** field, type `do400-monitoring-lab`.

**Note**

In the case where you are part of an organization, make sure that the repository is created under your Quay.io user.

- 4.4. Click **Public** in the **Repository Visibility** field. Do not change any other field.
- 4.5. Click **Create Public Repository**.
5. Store your GitHub and Quay.io credentials in Jenkins.
Create, and store a GitHub token to grant access to the private repository from the Jenkins pipeline. Name the GitHub token as `DO400_MONITORING_LAB` and

grant the `repo` permissions. Store the token in Jenkins in a credential with global scope, assign `MONITORING_LAB_GH_CREDENTIALS` to the credential ID, and assign `MONITORING_LAB_GH` to the credential description.

Store your Quay.io credentials in Jenkins. Use a credential with global scope, assign `MONITORING_LAB_QUAY_CREDENTIALS` to the credential ID, and assign `MONITORING_LAB_QUAY` to the credential description.

- 5.1. Return to the GitHub tab in your browser, and navigate to <https://github.com/settings/tokens>.
- 5.2. Click **Generate new token**. If prompted, enter your GitHub password. Type `DO400_MONITORING_LAB` in the **Note** field, select the `repo` check box in the list of scopes, and click **Generate token**.
After the token generation, copy the token value.
- 5.3. Return to the Jenkins tab in your browser. In the **Manage Jenkins** page, click **Manage Credentials** in the right pane.
- 5.4. Click **global** in the **Stores scoped to Jenkins** area, and then click **Add Credentials** in the left pane.
- 5.5. From the **Kind** list, select **Username with password**. From the **Scope** list, select **Global (Jenkins, nodes, items, all child items, etc)**. Type your GitHub user name in the **Username** field. Paste the GitHub token in the **Password** field. Type `MONITORING_LAB_GH_CREDENTIALS` in the **ID** field. Type `MONITORING_LAB_GH` in the **Description** field, and finally click **OK**.
- 5.6. Click **Add Credentials** in the left pane.
- 5.7. From the **Kind** list, select **Username with password**. From the **Scope** list, select **Global (Jenkins, nodes, items, all child items, etc)**. Type your Quay.io user name in the **Username** field. Type your Quay.io password in the **Password** field. Type `MONITORING_LAB_QUAY_CREDENTIALS` in the **ID** field. Type `MONITORING_LAB_QUAY` in the **Description** field, and finally click **OK**.
- 5.8. Click **Dashboard** in the breadcrumb trail to return to the Jenkins dashboard page.
6. Create a Jenkins project of the type **Pipeline** named `monitoring-lab`, and link it to the private GitHub repository. Use `*/main` as the branch specifier to only track changes in the main branch.
 - 6.1. In the left pane, click **New Item**.
 - 6.2. Assign the name `monitoring-lab` to the project, then click **Pipeline**, and finally click **OK**.
 - 6.3. Click the **Pipeline** tab. In the **Pipeline** area, from the **Definition** list, select **Pipeline script from SCM**.
 - 6.4. From the **SCM** list select **Git**. In the **Repository URL** type the URL of the private GitHub repository created in a preceding step.
The URL of the repository is similar to `https://github.com/YOUR_GITHUB_USER/DO400-monitoring-lab`.
 - 6.5. From the **Credentials** list, select the one that displays `MONITORING_LAB_GH`. In the **Branch Specifier** field, type `*/main`. In the **Script Path** field, type `Jenkinsfile`.

6.6. Click **Save** to save the pipeline and go to the pipeline details page.

7. Create a pipeline that uses the declarative syntax, and follows the pipeline as code technique. The calculator is a Quarkus application that requires the **maven** agent to execute the pipeline.

The pipeline must include the following stages to execute in sequential order:

- **Test**: executes the `./mvnw clean test` command to validate the application.
- **Build**: uses the `./scripts/include-container-extensions.sh` script to add the required dependencies, and the `./scripts/build-and-push-image.sh` script to build and push a container image with the application to Quay.io.

The `build-and-push-image.sh` script requires the following parameters:

- **-b**: the build number
- **-u**: your Quay.io user
- **-p**: your Quay.io password
- **-r**: the Quay.io repository

- 7.1. Create a file named `Jenkinsfile` in the root of the application folder to store the declarative pipeline. Add an agent section, and specify the use of a node with the label **maven**. Add a stage named **Test**, and include a step to execute the `./mvnw clean test` command.

The file contents should be similar to the following:

```
pipeline {
  agent { node { label 'maven' } }
  stages {
    stage('Test') {
      steps {
        sh './mvnw clean test'
      }
    }
  }
}
```

- 7.2. Add a stage named **Build** after the **Test** stage. This new stage has two steps. The first step adds the **kubernetes**, and **container-image-jib** extensions to the Quarkus application by using the `scripts/include-container-extensions.sh` script. The second step packages the application, creates a container image, and pushes the container image to your Quay.io repository. You must skip the tests in the second step to speed up the pipeline execution.

Use the `credentials` helper method to access your Quay.io credentials, and store the values in an environment variable.

The file contents should be similar to:

```
pipeline {
  agent { node { label 'maven' } }
  stages {
    stage('Tests') {
      ...output omitted...
    }
  }
}
```

```

stage('Build') {
    environment {
        QUAY = credentials('MONITORING_LAB_QUAY_CREDENTIALS')
    }
    steps {
        sh './scripts/include-container-extensions.sh'

        sh '''
            ./scripts/build-and-push-image.sh \
            -b $BUILD_NUMBER \
            -u "$QUAY_USR" \
            -p "$QUAY_PSW" \
            -r do400-monitoring-lab
        '''
    }
}
}

```

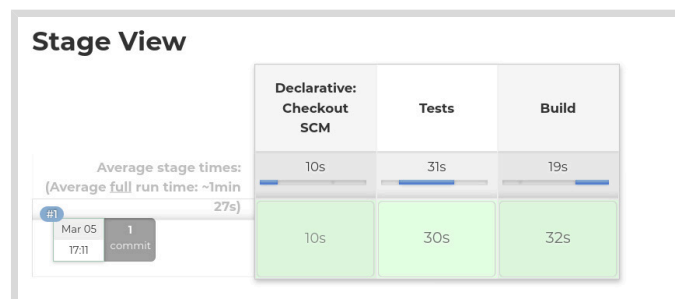
- 7.3. Commit and push the Jenkinsfile file to the remote repository.

```

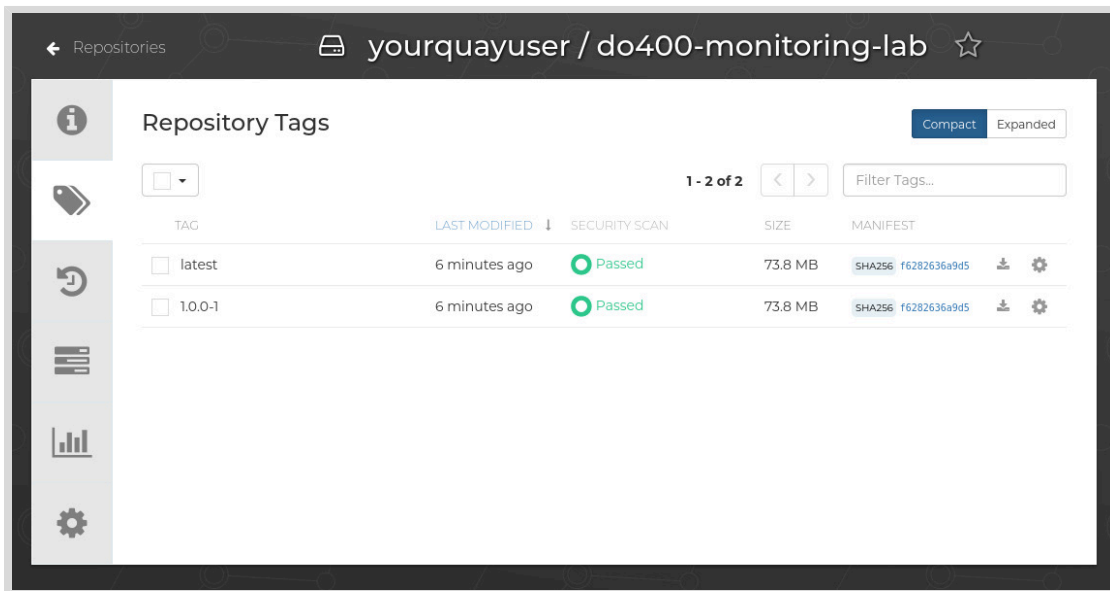
[user@host monitoring-lab]$ git add Jenkinsfile
[user@host monitoring-lab]$ git commit -m "Added Jenkins integration"
[main 77b5cd9] Added Jenkins integration
1 file changed, 26 insertions(+)
create mode 100644 Jenkinsfile
[user@host monitoring-lab]$ git push origin HEAD
...output omitted...
32ac625..77b5cd9 HEAD -> main

```

- 7.4. Return to the Jenkins tab in your browser. From the left side navigation, click **Build Now** to schedule a pipeline run. Wait for the build to finish and verify that it is successful.



- 7.5. Return to the Quay.io tab in your browser. Refresh the page, and click **Tags** to verify that the pipeline execution pushed a container image.



8. Configure Jenkins to send email notifications and alerts by using the SMTP server deployed in a preceding step. Capture pipeline failures, and send email alerts to `team@example.com`. Append the pipeline name to the email subject, and embed the build URL into the email body.

8.1. Return to the command-line terminal, and get the SMTP server route.

```
[user@host monitoring-lab]$ oc get route email \
-n RHT_OCP4_DEV_USER-monitoring-lab \
-o jsonpath="{ 'http://' }{ .spec.host }{ '\n' }"
http://email-your-user-notifications.apps.cluster.example.com
```

Open a new tab in the browser, and navigate to the URL that you just retrieved. You will use this web interface to view all the emails sent to the SMTP server.

- 8.2. Return to the Jenkins tab in your browser, and click **Dashboard** in the breadcrumb trail to return to the Jenkins dashboard page.
- 8.3. Click **Manage Jenkins** in the left pane, and then click **Configure System** in the right pane.
- 8.4. Scroll down to the **E-mail Notification** area to configure an SMTP server. Type `email.RHT_OCP4_DEV_USER-monitoring-lab` in the **SMTP server** field.
- 8.5. Select the **Test configuration by sending test e-mail** check box, and then type `test@example.com` in the **Test e-mail recipient** field. Click **Test configuration**, and a success message displays to confirm that the testing email was sent. Click **Save**.
- 8.6. Return to the SMTP web interface tab in your browser. Verify that the web interface displays a test email sent to `test@example.com` in the left pane, and click **Clear Inbox** to remove all the stored emails.
- 8.7. Add a **post** section with a **failure** condition in the pipeline. The file contents should be similar to:

```

pipeline {
  agent { node { label 'maven' } }
  stages {
    ...output omitted...
  }
  post {
    failure {

    }
  }
}

```

- 8.8. Add a `mail` step in the `failure` condition to send an alert to `team@example.com`. Append the pipeline name to the email subject, and embed the build URL into the email body. The file contents should be similar to:

```

pipeline {
  ...output omitted...
  post {
    failure {
      mail to: "team@example.com",
           subject: "Pipeline failed: ${currentBuild.fullDisplayName}",
           body: "The following pipeline failed: ${env.BUILD_URL}"
    }
  }
}

```

- 8.9. Commit and push the changes made to the `Jenkinsfile` file to the remote repository.

```

[user@host monitoring-lab]$ git add Jenkinsfile
[user@host monitoring-lab]$ git commit -m "Added alerts"
[main 77b5cd9] Added alerts
 1 file changed, 7 insertions(+)
[user@host monitoring-lab]$ git push origin HEAD
...output omitted...
32ac625..77b5cd9 HEAD -> main

```

9. Deploy the calculator application manually by using the `kubefiles/application-template.yml` template. Add a stage named `Deploy` that deploys the latest container image available in Red Hat OpenShift. Use the `scripts/redeploy.sh` script for deployments executed in the pipeline.

The `redeploy.sh` script requires the following parameters:

- `-d`: deployment name
- `-n`: namespace to use

- 9.1. Use the `oc` command to deploy the application.

```
[user@host monitoring-lab]$ oc process \
-n RHT_OCP4_DEV_USER-monitoring-lab \
-f kubefiles/application-template.yml \
-p QUAY_USER_OR_GROUP=YOUR_QUAY_USER \
-p QUAY_REPOSITORY=do400-monitoring-lab \
| oc apply -n RHT_OCP4_DEV_USER-monitoring-lab -f -
deployment.apps/calculator created
service/calculator created
route.route.openshift.io/calculator created
```

Notice that you must replace the `YOUR_QUAY_USER` string with your Quay username.

- 9.2. Get the URL of the application to verify the successful deployment of the calculator. Append `/solver/4.02+8.0-1.1` to the URL to verify that the application is running.

```
[user@host monitoring-lab]$ oc get route calculator \
-n RHT_OCP4_DEV_USER-monitoring-lab \
-o jsonpath="{ 'http://'{ .spec.host } '/solver/4.02+8.0-1.1' }{'\n'}"
http://calculator-your-user-monitoring-lab.apps.cluster.example.com/
solver/4.02+8.0-1.1
```

Open a new browser tab and navigate to the applications URL to check that it is working correctly. Verify that the application returns the `10.92` response. You might need to wait a few seconds until the application is completely deployed.

- 9.3. Add a stage named `Deploy` after the `Build` stage to deploy new versions of the calculator to Red Hat OpenShift. The file contents should be similar to:

```
pipeline {
  agent { node { label 'maven' } }
  stages {
    ...output omitted...
    stage('Build') {
      ...output omitted...
    }
    stage('Deploy') {
      steps {
        sh '''
          ./scripts/redeploy.sh \
            -d calculator \
            -n RHT_OCP4_DEV_USER-monitoring-lab
          ...
        '''
      }
    }
  }
  ...output omitted...
}
```

Notice that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user.

- 9.4. Commit and push the changes made to the `Jenkinsfile` file to the remote repository.

```
[user@host monitoring-lab]$ git add Jenkinsfile
[user@host monitoring-lab]$ git commit -m "Added Deploy stage"
[main 5953194] Added Deploy stage
 1 file changed, 9 insertions(+)
[user@host monitoring-lab]$ git push origin HEAD
...output omitted...
32ac625..77b5cd9 HEAD -> main
```

- 9.5. Return to the Jenkins tab in your browser. Click **monitoring-lab** in the right pane, and then click **Build Now** to run the pipeline. Wait for the build to finish and verify that it is successful.
- 9.6. Return to the application tab in your browser. Refresh the page and verify that the application continues working.
10. Add security stage to scan the calculator container image for vulnerabilities. Name this stage **Security Scan**, and add it to the pipeline between the **Build** and **Deploy** stages.
Use the `kubefiles/security-scan-template.yml` template in combination with the `oc replace --force` command to create the Job that scans the container image.
Examine the template to find out the required parameters. Use the `scripts/check-job-state.sh` script to check the status of the scan job.

**Note**

Security experts find new vulnerabilities every day. This exercise only looks for one of them, CVE-2021-23840, to ensure that the result of the exercise remains deterministic.

In a real world scenario, you should check for all the vulnerabilities.

- 10.1. Open the `Jenkinsfile` file and add a stage named **Security** between the **Build** and **Deploy** stages.

The file contents should be similar to:

```
pipeline {
  agent { node { label 'maven' } }
  stages {
    stage('Tests') {
      ...output omitted...
    }
    stage('Build') {
      ...output omitted...
    }
    stage('Security Scan') {
      steps {
        sh '''
          oc process -f kubefiles/security-scan-template.yml \
            -n RHT_OCP4_DEV_USER-monitoring-lab \
            -p QUAY_USER=YOUR_QUAY_USER \
            -p QUAY_REPOSITORY=do400-monitoring-lab \
            -p APP_NAME=calculator \
            -p CVE_CODE=CVE-2021-23840 \
```

```

        | oc replace --force \
        -n RHT_OCP4_DEV_USER-monitoring-lab -f -
    ...
    sh '''
    ./scripts/check-job-state.sh "calculator-trivy" \
    "RHT_OCP4_DEV_USER-monitoring-lab"
    ...
    }
}
stage('Deploy') {
    ...output omitted...
}
}
...output omitted...
}

```

Notice that you must replace the `RHT_OCP4_DEV_USER` string with your OpenShift developer user, and the `YOUR_QUAY_USER` string with your Quay username.

- 10.2. Commit and push the changes made to the `Jenkinsfile` file to the remote repository.

```

[user@host monitoring-lab]$ git add Jenkinsfile
[user@host monitoring-lab]$ git commit -m "Added Scan stage"
[main a5cb3cb] Added Scan stage
1 file changed, 13 insertions(+)
[user@host monitoring-lab]$ git push origin HEAD
...output omitted...
32ac625..77b5cd9 HEAD -> main

```

- 10.3. Return to the Jenkins tab in your browser, and click **Build Now** to run the security scan. Wait for the pipeline execution to finish and verify that it fails in the **Security Scan** stage.
- 10.4. Return to the command-line terminal and verify the output of the security scan job.

```

[user@host monitoring-lab]$ oc logs job/calculator-trivy \
-n RHT_OCP4_DEV_USER-monitoring-lab
| libcrypto1.1 | CVE-2021-23840 ...output omitted...
| libssl1.1    | CVE-2021-23840 ...output omitted...

```



Note

We are using an old base image in this lab, and the number of vulnerabilities might differ from the example.

- 10.5. Return to the SMTP web interface tab in your browser to verify the alert integration. Verify that the web interface displays an email sent to `team@example.com` in the left pane, and click **Clear Inbox** to remove all the stored emails.
- 10.6. Change the base image for the application to one that is more up to date. Open the `scripts/build-and-push-image.sh` script and add the `quarkus.jib.base-jvm-image` variable. Set the value of the variable to

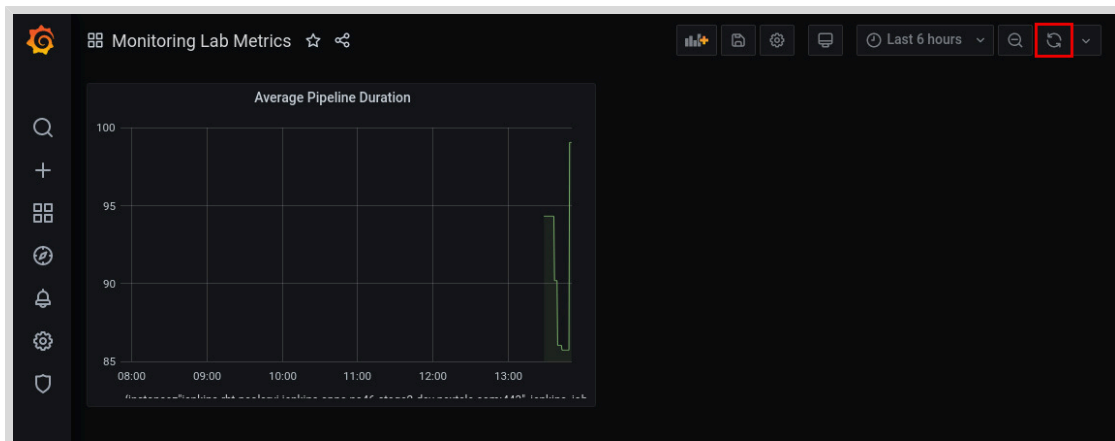
registry.access.redhat.com/ubi8/openjdk-11:1.3-10 to define the base image.

```
...output omitted...
./mvnw package -DskipTests \
-Dquarkus.jib.base-jvm-image=registry.access.redhat.com/ubi8/openjdk-11:1.3-10 \
-Dquarkus.container-image.build=true \
-Dquarkus.container-image.registry=quay.io \
...output omitted...
```

- 10.7. Commit and push the changes made to the scripts/build-and-push-image.sh file to the remote repository.

```
[user@host monitoring-lab]$ git add scripts/build-and-push-image.sh
[user@host monitoring-lab]$ git commit -m "Changed base image"
[main beb7e9d] Changed base image
1 file changed, 1 insertion(+)
[user@host monitoring-lab]$ git push origin HEAD
...output omitted...
32ac625..77b5cd9 HEAD -> main
```

- 10.8. Return to the Jenkins tab in your browser, and then click **Build Now** to run the security scan. Wait for the build to finish and verify that it is successful.
- 10.9. Return to the Grafana tab and verify that the **Average Pipeline Duration** was updated. Click the refresh button to update the metric graphic.



11. Clean up. Delete the Jenkins project, and the OpenShift project.
 - 11.1. Return to the Jenkins tab in your browser, and then click **Delete Pipeline** in the left pane. A dialog displays requesting confirmation. Click **OK** to delete the pipeline.
 - 11.2. Return to the command-line terminal and delete the lab project.

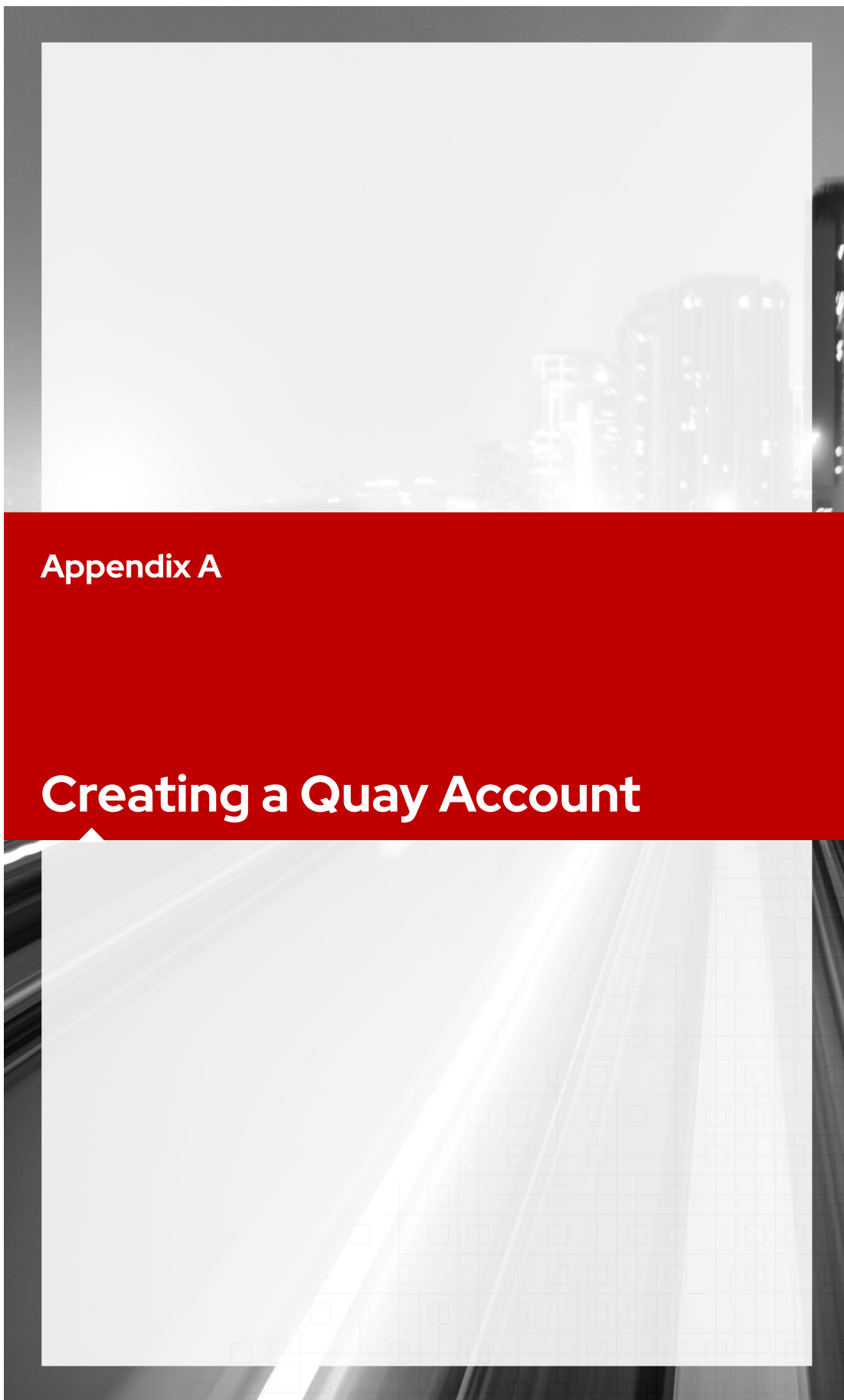
```
[user@host monitoring-lab]$ oc delete project \
RHT_OCP4_DEV_USER-monitoring-lab
project.project.openshift.io "your-user-monitoring-lab" deleted
```

This concludes the lab.

Summary

In this chapter, you learned:

- DevSecOps aims at integrating security across the whole software development process.
- Security is any measure taken to prevent service disruption and data leakage or loss.
- By building security into your pipelines, you can make sure your containers are trusted and reliable.
- By capturing metrics around the software delivery process, you can analyze the improvements in the adoption of DevOps practices.
- An important part of the DevOps mindset is the continuous monitoring of your software life cycle to discover problems and react to them faster.



Creating a Quay Account

Objectives

After completing this section, you should be able to describe how to create a Quay account and public container image repositories for labs in the course

Creating a Quay Account

You need a Quay account to create one or more *public* container image repositories for the labs in this course. If you already have a Quay account, then you can skip the steps to create a new account listed in this appendix.



Important

If you already have a Quay account, then ensure that you only create *public* container image repositories for the labs in this course. The lab grading scripts and instructions require unauthenticated access to pull container images from the repository.

To create a new Quay account, perform the following steps:

1. Navigate to <https://quay.io> by using a web browser.
2. Click **Sign in** in the upper-right corner (next to the search bar).
3. On the **Sign in** page, you can log in using your Red Hat, Google, or GitHub credentials.

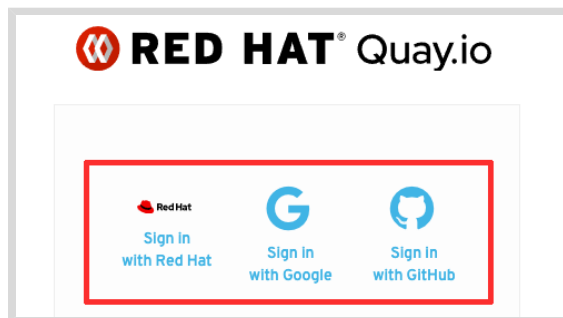


Figure A.1: Sign in using Red Hat, Google, or GitHub credentials

Alternatively, click **Create Account** to create a new account.

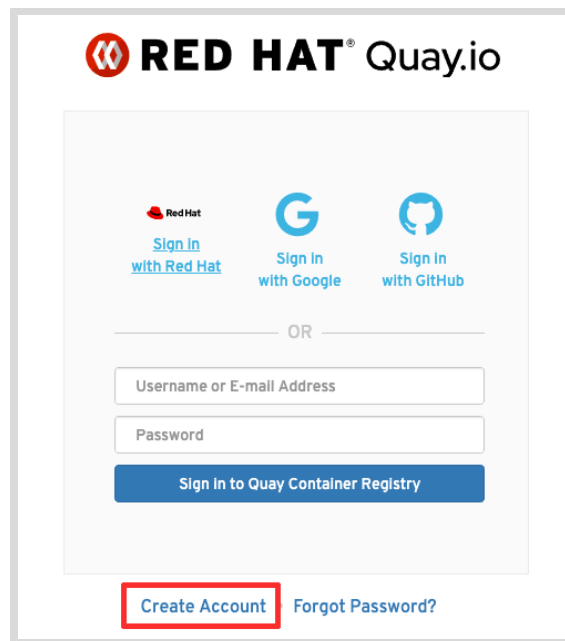


Figure A.2: Creating a new account

4. If you chose to skip the Red Hat, Google, or GitHub login method and instead opt to create a new account, then you will receive an email with instructions on how to activate your Quay account. Verify your email address and then sign in to the Quay website with the username and password you provided during account creation.
5. After you have logged in to Quay you can create new image repositories by clicking **Create New Repository** on the Repositories page.

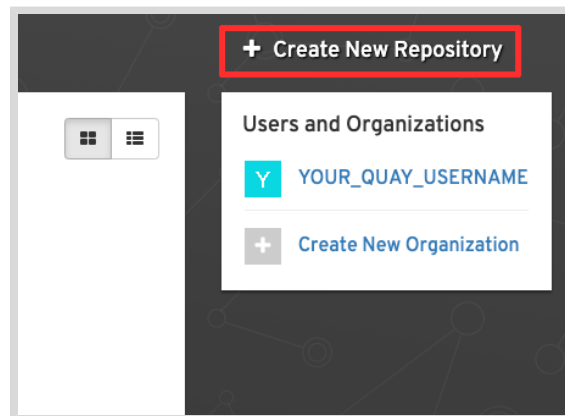


Figure A.3: Creating a new image repository

Alternatively, click the plus icon (+) in the upper-right corner (to the left of the bell icon), and then click **New Repository**.

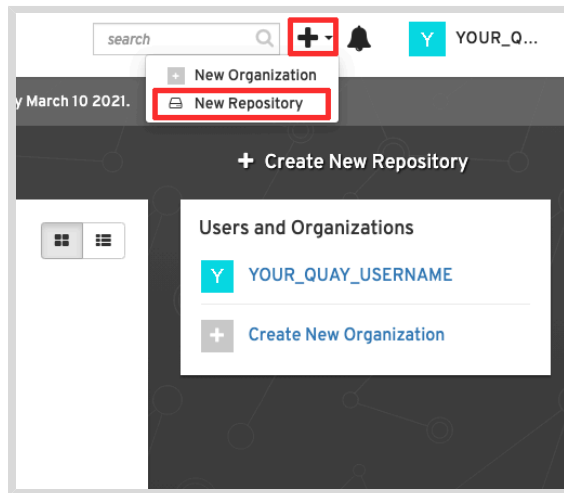


Figure A.4: Creating a new image repository

6. Enter a name for the repository as per your lab instructions. Ensure that you select the **Public**, and **Empty Repository** options.

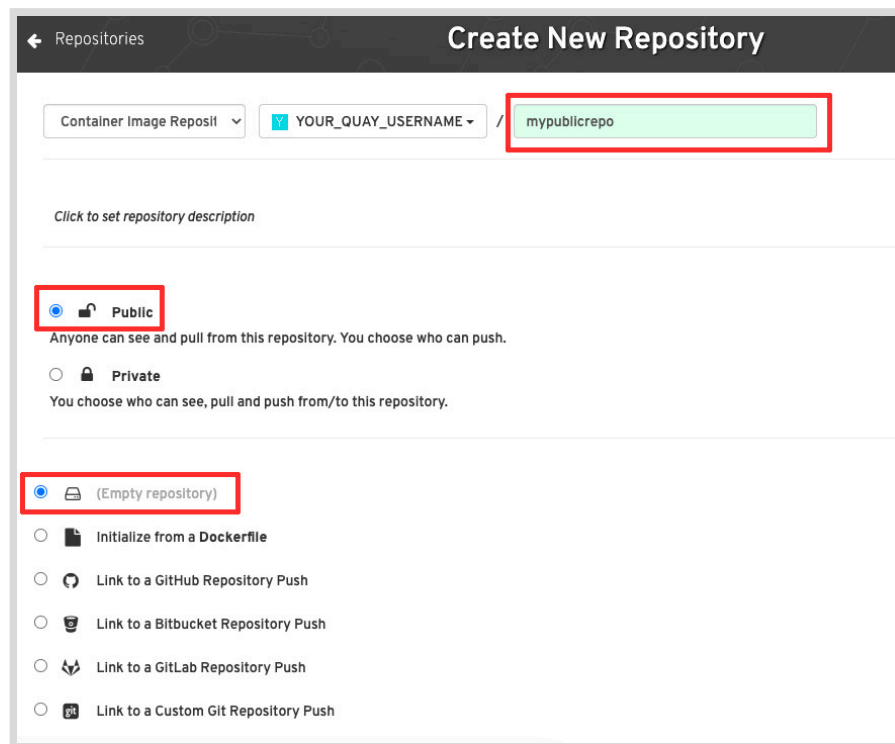


Figure A.5: Creating a new image repository

Click **Create Public Repository** to create the repository.



References

Getting Started with Quay.io

<https://docs.quay.io/solution/getting-started.html>