Red Hat
Training and Certification

Student Workbook (ROLE)

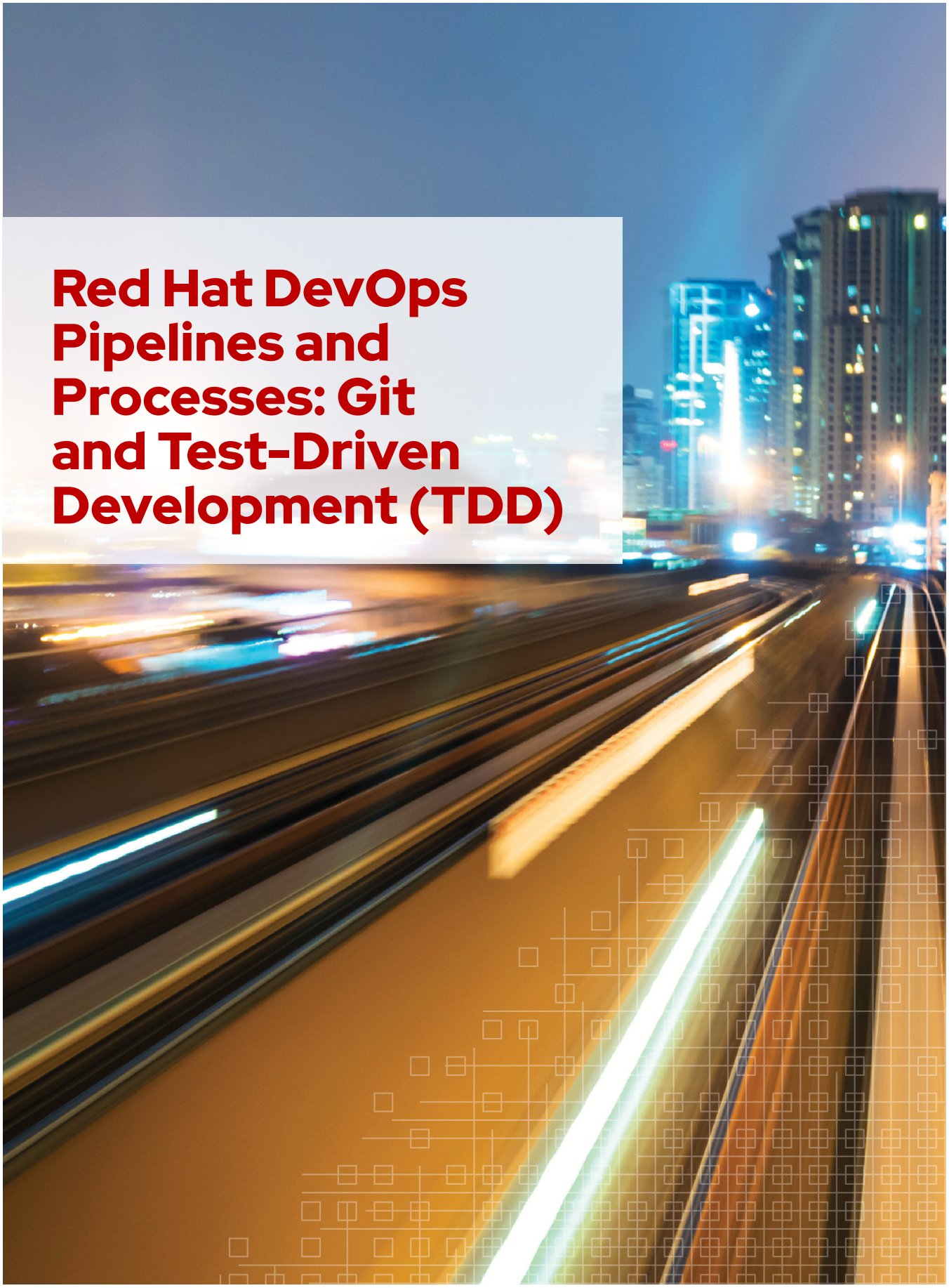
OCP 4.6 DO402

Red Hat DevOps Pipelines and Processes: Git and Test-Driven Development (TDD)

Edition 4

DO402-OCP4.6-en-4-20210827

Red Hat DevOps Pipelines and Processes: Git and Test-Driven Development (TDD)



OCP 4.6 DO402
Red Hat DevOps Pipelines and Processes: Git and Test-Driven Development (TDD)
Edition 4 20210827
Publication date 20210827

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

Editor: Sam Ffrench

Contributors: David Sacco, Richard Allred, Zachary Gutterman

Document Conventions

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

Admonitions

Red Hat Training courses use the following admonitions:

References
References describe where to find external documentation that is relevant to a subject.

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

Warning
Warnings 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 that are covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Red Hat DevOps Pipelines and Processes: Git and Test-Driven Development (TDD)

Red Hat DevOps Pipelines and Processes: Git and Test Driven-Development (TDD) (DO402) is designed for developers to learn version control, test-driven development, Jenkins, and other tools that support a DevOps organization.

Course Objectives

- Students will be able to use the concepts in this course to simplify and more efficiently integrate their application code, build reliable code with TDD, and use automated pipelines to simplify their testing. Students will learn how to design applications with a "test first" approach, integrate application code with Git, and leverage Jenkins pipelines for automated deployment and testing. This course is intended to illustrate the benefits of DevOps and the tools that support its implementation.

Audience

- This course is intended for application developers.

Prerequisites

- The course Red Hat Application Development I: Programming in Java EE (JB183), or experience with application development is strongly recommended, but not required. Be proficient in using an IDE such as Red Hat® Developer Studio or VSCode.

Orientation to the Classroom Environment

DO402 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

Attribute	*Minimum Requirements*	*Recommended*
CPU	1.6 GHz or faster processor	Multi-core i7 or equivalient
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 DO402 provide instructions to install the following programs:

- Python 3
- Node.js
- JDK
- Git 2.18 or later (Git Bash for Windows systems)

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

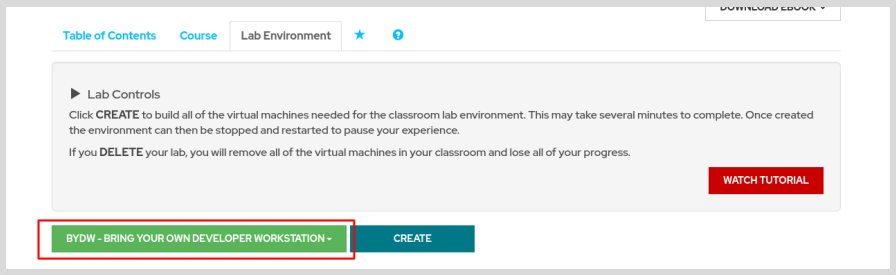
- You might find differences in the prompt format.
- In macOS, your prompt might be similar to `host:~ user$`.
- When performing the exercises, interpret the `[user@host ~]$` prompt used in the course as a representation of your macOS prompt.
- All the commands from the exercises should be functional.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

Microsoft Windows

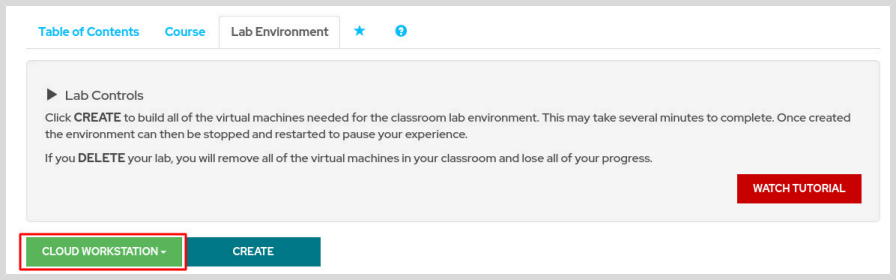
- Windows does not support Bash natively. Instead, you must use PowerShell.
- In Windows PowerShell, your prompt should be similar to `PS C:\Users\user>`.
- When performing the exercises, interpret the `[user@host ~]$` Bash prompt as a representation of your Windows PowerShell prompt.
- For some commands, 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

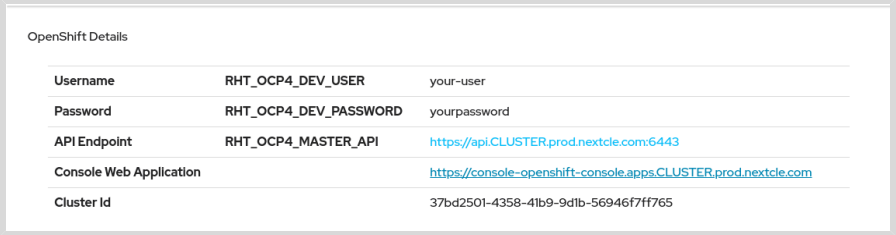
To use your own system or an ILT workstation provided by your instructor, select the **BYDW** option from the dropdown when you launch your classroom by using the **CREATE** button in the **Lab Environment** tab in the Red Hat Online Learning (ROL) interface.



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.



The required tools are pre-installed in the `Cloud Workstation` classroom environment, which also includes VSCodium, 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.

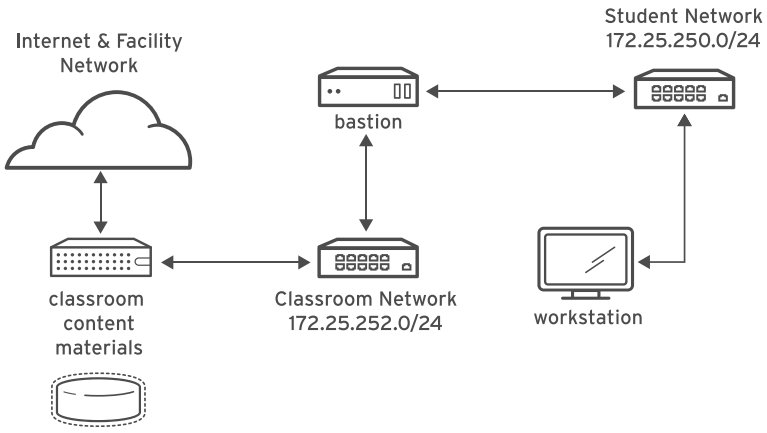


Figure 0.4: Cloud workstation classroom overview

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
workstation.lab.example.com	172.25.250.9	Graphical workstation used by students
bastion.lab.example.com	172.25.250.254	Router linking student's VMs to classroom servers
classroom.lab.example.com	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. Self-paced courses are accessed through a web application that is hosted at rol.redhat.com [http://rol.redhat.com]. If your course is an instructor-led virtual training event, you will be provided with your course location URL. Log in to this site with your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

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

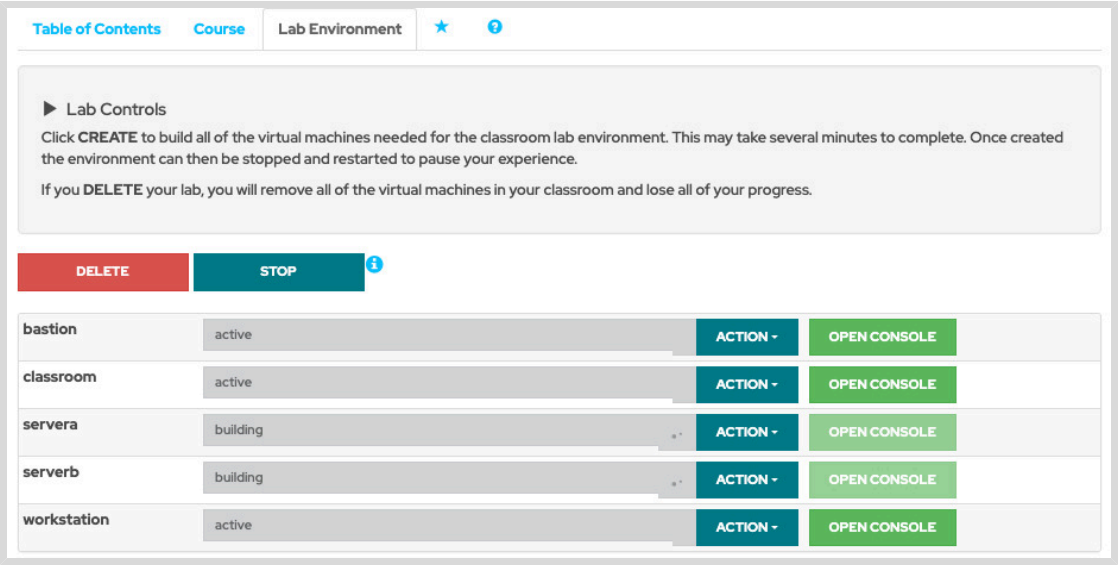


Figure 0.5: An example course Lab Environment management page

Machine States

Virtual Machine State	Description
building	The virtual machine is being created.
active	The virtual machine is running and available. If just started, it might still be starting services.
stopped	The virtual machine is completely shut down. On starting, the virtual machine boots into the same state as it was before it was shut down. The disk state is preserved.

Classroom Actions

Button or Action	Description
CREATE	Create the ROLE classroom. Creates and starts all of the virtual machines that are needed for this classroom.
CREATING	The ROLE classroom virtual machines are being created. Creates and starts all of the virtual machines that are needed for this classroom. Creation can take several minutes to complete.
DELETE	Delete the ROLE classroom. Destroys all virtual machines in the classroom. **All saved work on that system's disks is lost.**
START	Start all virtual machines in the classroom.
STARTING	All virtual machines in the classroom are starting.
STOP	Stop all virtual machines in the classroom.

Machine Actions

Button or Action	Description
OPEN CONSOLE	Connect to the system console of the virtual machine in a new browser tab. You can log in directly to the virtual machine and run commands, when required. Normally, log in to the `workstation` virtual machine only, and from there, 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 disk contents.
ACTION → Power Off	Forcefully shut down the virtual machine, while still preserving disk contents. 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. **All saved work on that system's disks 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** on every virtual machine in the list.

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

Warning
The DELETE operation cannot be undone. All completed work in the classroom environment is lost.

The Auto-stop and Auto-destroy Timers

The Red Hat Online Learning enrollment entitles you to a set allotment of computer time. To help to conserve your allotted time, the ROLE classroom uses timers, which shut down or delete the classroom when the appropriate timer expires.

To adjust the timers, locate the two + buttons at the bottom of the course management page. Click the auto-stop + button to add another hour to the auto-stop timer. Click the auto-destroy + button to add another day to the auto-destroy timer. Auto-stop has a maximum of 11 hours, and auto-destroy has a maximum of 14 days. Be careful to keep the timers set while you are working, so that your environment is not unexpectedly shut down. Be careful not to set the timers unnecessarily high, which could waste your subscription time allotment.

Performing Lab Exercises

Run the `lab` command from `workstation` to prepare your environment before each hands-on exercise, and again to clean up after an exercise. Each hands-on exercise has a unique name within a course. The exercise is prepended with `lab-` as its file name in `/usr/local/lib`. For example, the `instances-cli` exercise has the file name `/usr/local/lib/lab-instances-cli`. To list the available exercises, use tab completion in the `lab` command:

```
[student@workstation ~]$ lab Tab Tab
administer-users   deploy-overcloud-lab  prep-deploy-ips         stacks-autoscale
analyze-metrics    instances-cli         prep-deploy-router      stacks-deploy
assign-roles       manage-interfaces     public-instance-deploy  verify-overcloud
```

Exercises are in two types. The first type, a *guided exercise*, is a practice exercise that follows a course narrative. If a narrative is followed by a quiz, then usually the topic did not have an achievable practice exercise. The second type, an *end-of-chapter lab*, is a gradable exercise to help verify your learning. When a course includes a comprehensive review, the review exercises are structured as gradable labs. The syntax for running an exercise script is:

```
[student@workstation ~]$ lab exercise action
```

The *action* is a choice of `start`, `grade`, or `finish`. All exercises support `start` and `finish`. Only end-of-chapter labs and comprehesive review labs support `grade`. Older courses might still use `setup` and `cleanup` instead of the current `start` and `finish` actions.

start
: Formerly `setup`. A script's start logic verifies the required resources to begin an exercise. It might include configuring settings, creating resources, checking prerequisite services, and verifying necessary outcomes from previous exercises. You can take an exercise at any time, even without taking prerequisite exercises.

grade
: End-of-chapter labs help verify what you have learned, after practicing with earlier guided exercises. The grade action directs the `lab` command to display a list of grading criteria, with a PASS or FAIL status for each. To achieve a PASS status for all criteria, fix the failures and rerun the grade action.

finish
: Formerly `cleanup`. A script's finish logic deletes exercise resources that are no longer necessary. You can take an exercise as many times as you want.

Troubleshooting Lab Scripts

Exercise scripts do not exist on `workstation` until each is first run. When you run the `lab` command with a valid exercise and action, the script named `lab-exercise` is downloaded from the `classroom` server content share to `/usr/local/lib` on `workstation`. The `lab` command creates two log files in `/var/tmp/labs`, plus the directory if it does not exist. One file, named *`exercise`*, captures standard output messages that are normally displayed on your terminal. The other file, named *`exercise.err`*, captures error messages.

```
[student@workstation ~]$ ls -l /usr/local/lib
-rwxr-xr-x. 1 root root  4131 May  9 23:38 lab-instances-cli
-rwxr-xr-x. 1 root root 93461 May  9 23:38 labtool.cl110.shlib
-rwxr-xr-x. 1 root root 10372 May  9 23:38 labtool.shlib

[student@workstation ~]$ ls -l /var/tmp/labs
-rw-r--r--. 1 root root   113 May  9 23:38 instances-cli
-rw-r--r--. 1 root root   113 May  9 23:38 instances-cli.err
```

Note

Scripts download from the `http://content.example.com/courses/course/release/grading-scripts` share, but only if the script does not yet exist on `workstation`. When you need to download a script again, such as when a script on the share is modified, manually delete the current exercise script from `/usr/local/lib` on `workstation`, and then run the `lab` command for the exercise again. The newer exercise script then downloads from the `grading-scripts` share.

To delete all current exercise scripts on `workstation`, use the `lab` command's `--refresh` option. A refresh deletes all scripts in `/usr/local/lib` but does not delete the log files.

```
[student@workstation ~]$ lab --refresh
[student@workstation ~]$ ls -l /usr/local/lib

[student@workstation ~]$ ls -l /var/tmp/labs
-rw-r--r--. 1 root root   113 May  9 23:38 instances-cli
-rw-r--r--. 1 root root   113 May  9 23:38 instances-cli.err
```

Interpreting the Exercise Log Files

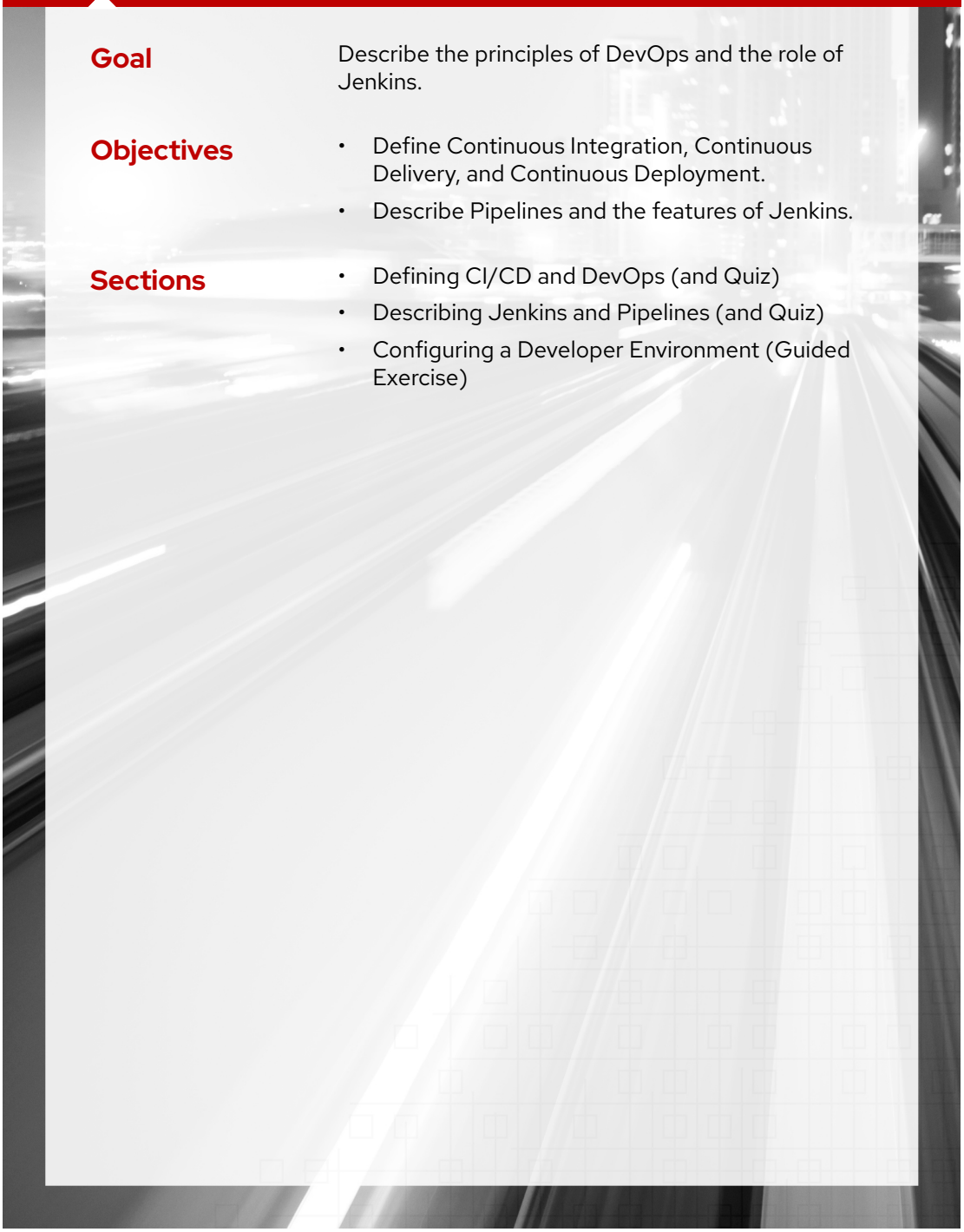
Exercise scripts send output to log files, even when the scripts are successful. Step header text is added between steps, and additional date and time headers are added at the start of each script run. The exercise log normally contains messages that indicate successful completion of command steps. Therefore, the exercise output log is useful for observing messages that are expected if no problems occur, but offers no additional help when failures occur.

Instead, the exercise error log is more useful for troubleshooting. Even when the scripts succeed, messages are still sent to the exercise error log. For example, a script that verifies that an object already exists before attempting to create it should cause an *object not found* message when the object does not exist yet. In this scenario, that message is expected and does not indicate a failure. Actual failure messages are typically more verbose, and experienced system administrators should recognize common log message entries.

Although exercise scripts are always run from `workstation`, they perform tasks on other systems in the course environment. Many course environments, including OpenStack and OpenShift, use a command-line interface (CLI) that is invoked from `workstation` to communicate with server systems by using API calls. Because script actions typically distribute tasks to multiple systems, additional troubleshooting is necessary to determine where a failed task occurred. Log in to those other systems and use Linux diagnostic skills to read local system log files and determine the root cause of the lab script failure.

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.

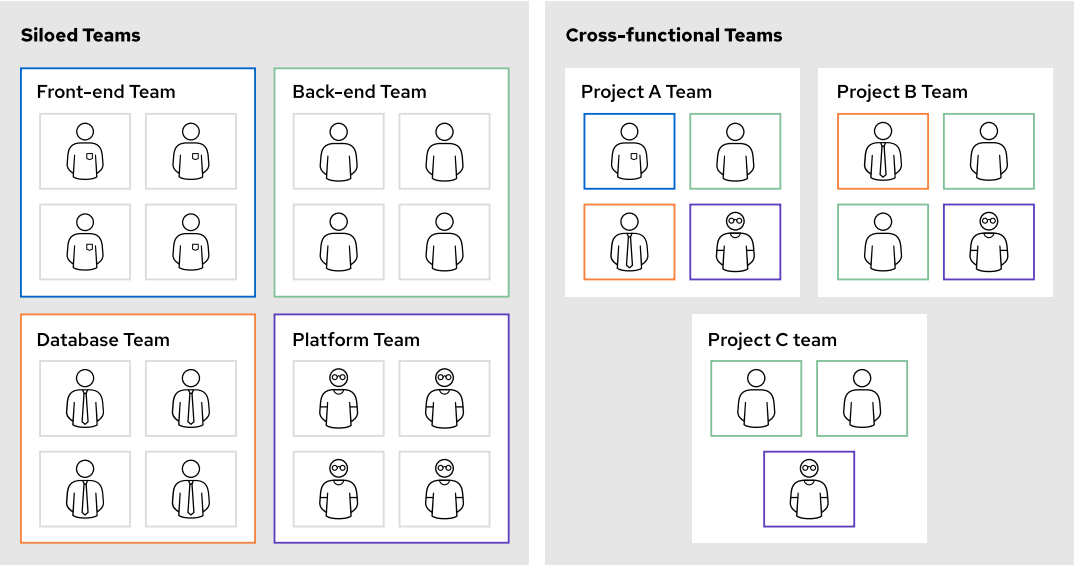


Figure 1.1: Cross-functional teams break down knowledge silos

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.

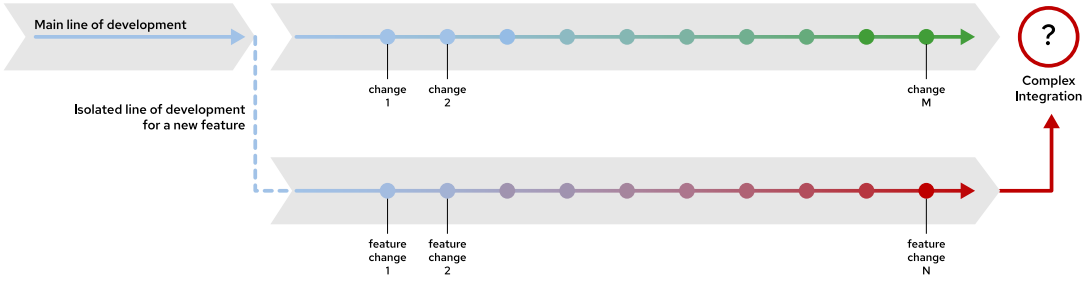


Figure 1.2: Long-lived isolated branches are difficult to integrate into the main branch

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.

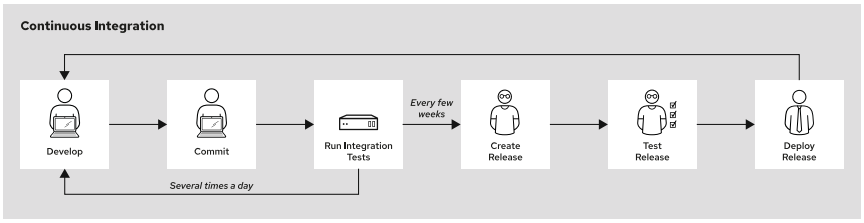


Figure 1.3: Continuous Integration

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.

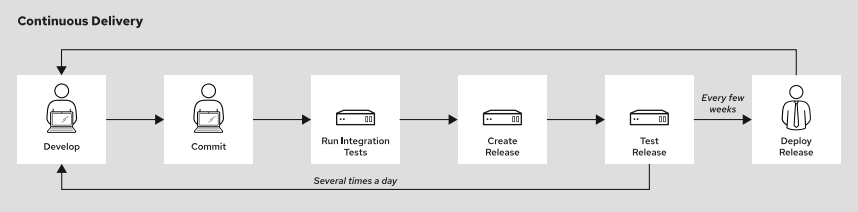


Figure 1.4: Continuous Delivery

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.

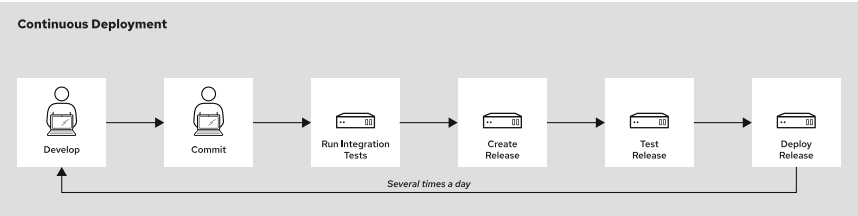


Figure 1.5: Continuous Deployment

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.

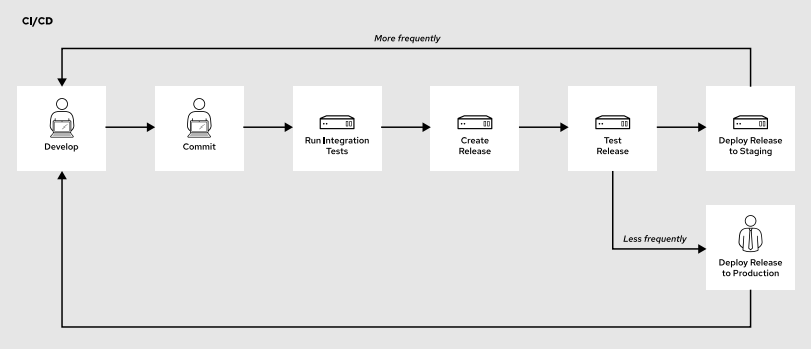


Figure 1.6: Example of CI/CD: Continuous Deployment for the staging environment. Continuous Delivery for the production environment

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.

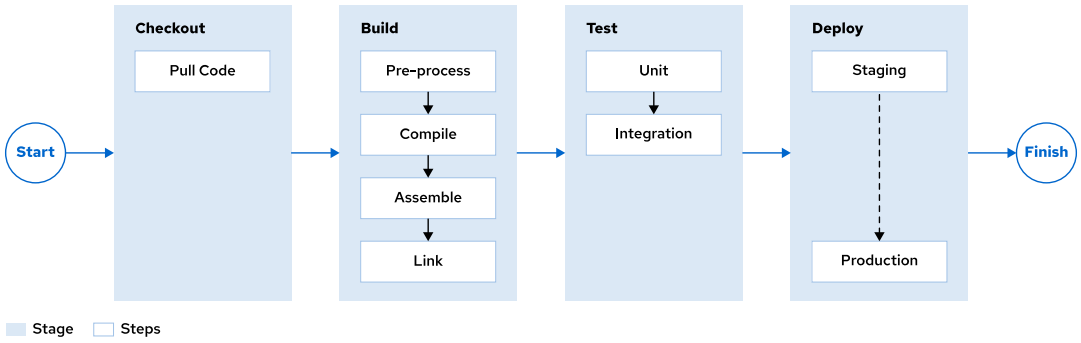


Figure 1.7: Execution flow example of a pipeline

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*.

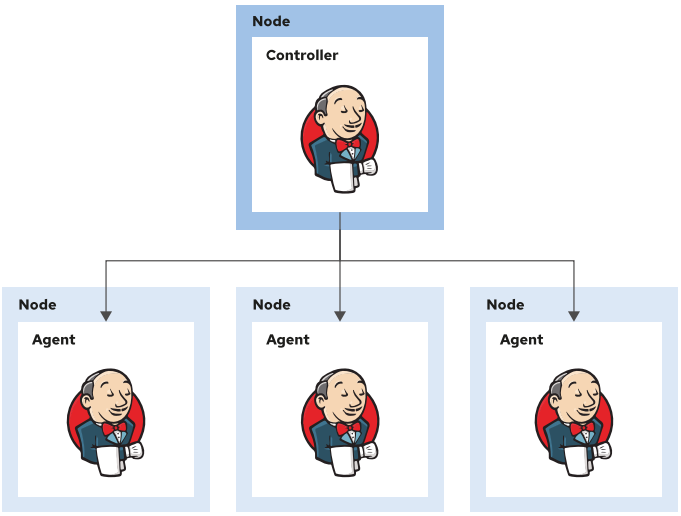


Figure 1.8: Jenkins architecture

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 \
/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.

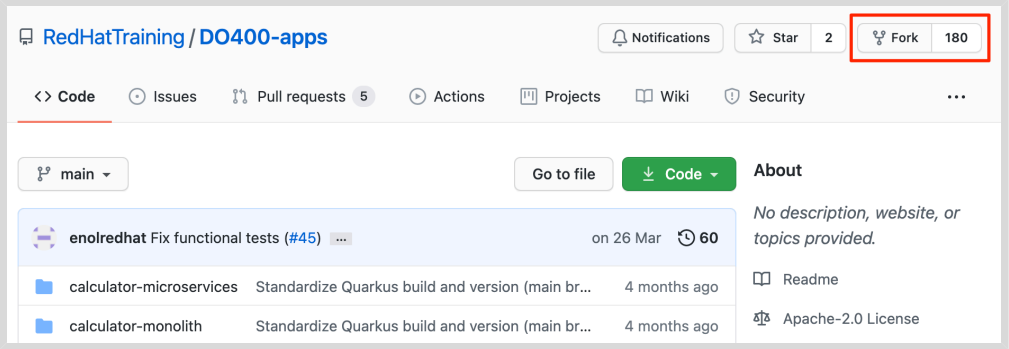
```
PS C:\Users\user> git --version
git version 2.30.0.windows.1
```

5. For some of the exercises, you need the sample applications of this course. Fork the `DO400-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 `~/DO400` workspace folder but you can choose a different one.

```
[user@host ~]$ mkdir DO400
[user@host ~]$ cd DO400
[user@host DO400]$
```

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

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

```
[user@host DO400]$ cd DO400-apps
[user@host DO400-apps]$ head README.md
# DO400 Application Repository
...output omitted...
[user@host DO400-apps]$ cd ..
[user@host DO400]$
```

Important

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

```
PS C:\Users\user\DO400\DO400-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] 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.

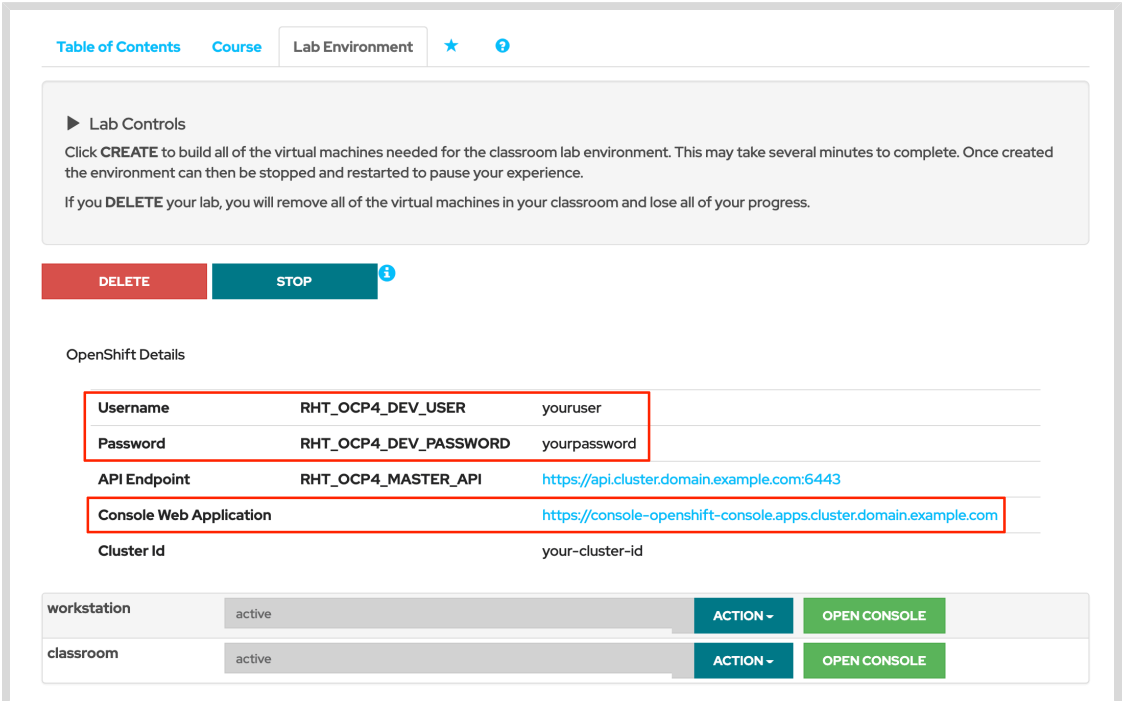
Note

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.



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
```

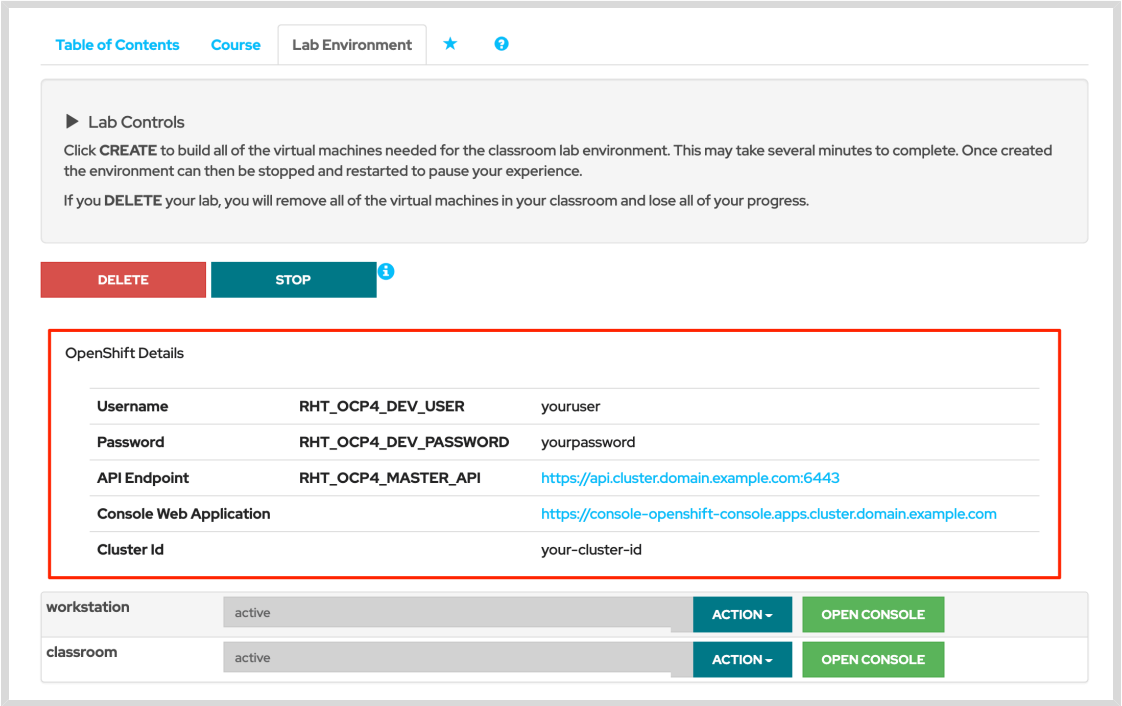
Note

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:



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 backlash 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 tenent 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 `~/DO400`. 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 DO400]$ git config --global user.name "Your User Name"
```

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

```
[user@host DO400]$ git config --global user.email your@user.email
```

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

```
[user@host DO400]$ 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 DO400]$ mkdir git-basics
```

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

```
[user@host DO400]$ 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.../DO400/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 ~/DO400
[user@host DO400]$
```

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.

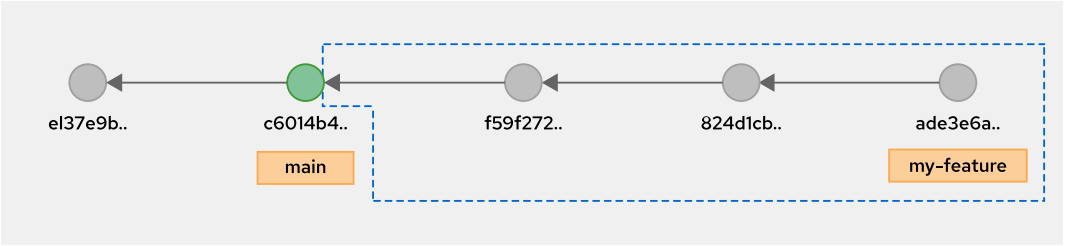


Figure 2.2: Commits in a branch not yet ready for the main branch

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:

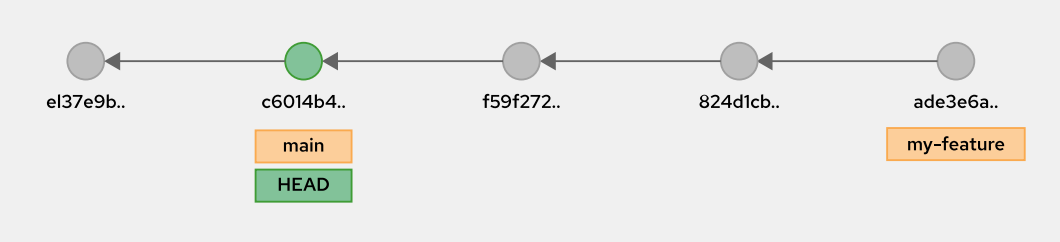


Figure 2.3: Example repository structure

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:

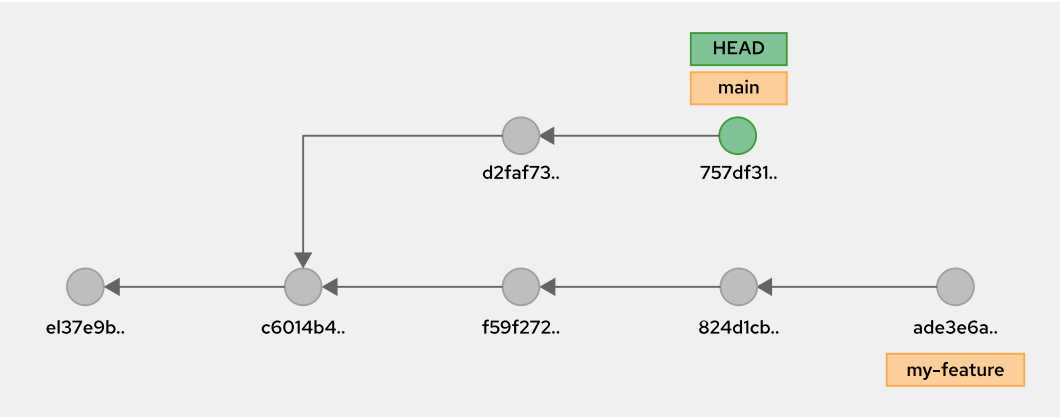


Figure 2.5: Example diverging repository structure

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:

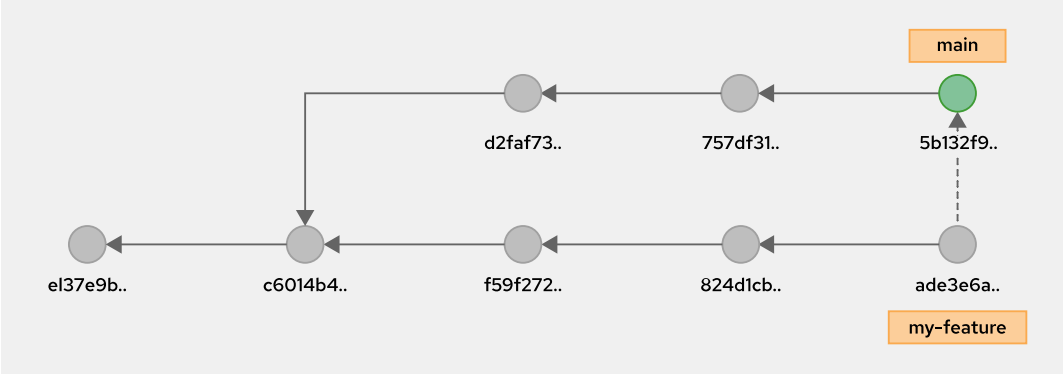


Figure 2.6: Example merge resulting in a merge commit

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 `~/DO400`. 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 DO400]$ mkdir hello-branches
```

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

```
[user@host DO400]$ cd hello-branches
[user@host hello-branches]$ git init .
Initialized empty Git repository in ...output omitted.../DO400/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 daf5db416ffd47ab6d95a16eaaf4dabbcfa2cb72 (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 daf5db416ffd47ab6d95a16eaaf4dabbcfa2cb72 (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 daf5db416ffd47ab6d95a16eaaf4dabbcfa2cb72
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.

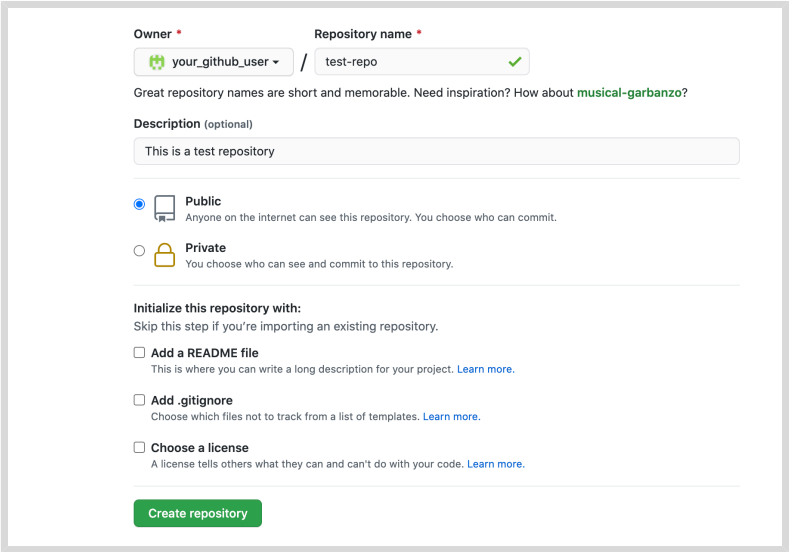


Figure 2.7: GitHub new repository form

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:

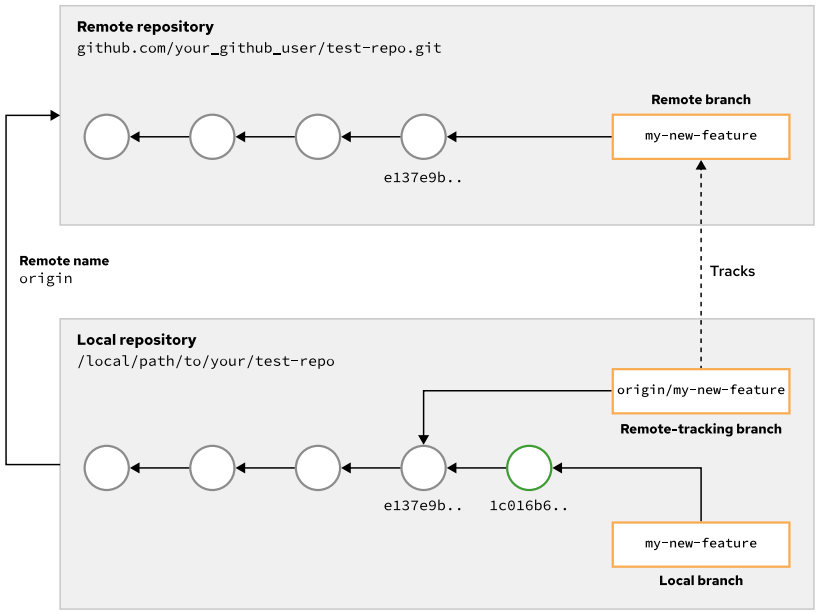


Figure 2.8: The local my-new-feature branch is ahead of the origin/ my-new-feature branch and the remote my-new-feature branch

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 (1)
...output omitted...
[user@host your-local-repo]$ git push origin my-new-feature (2)
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.

(1) Check that the currently selected local branch is `my-new-feature`.

(2) 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.

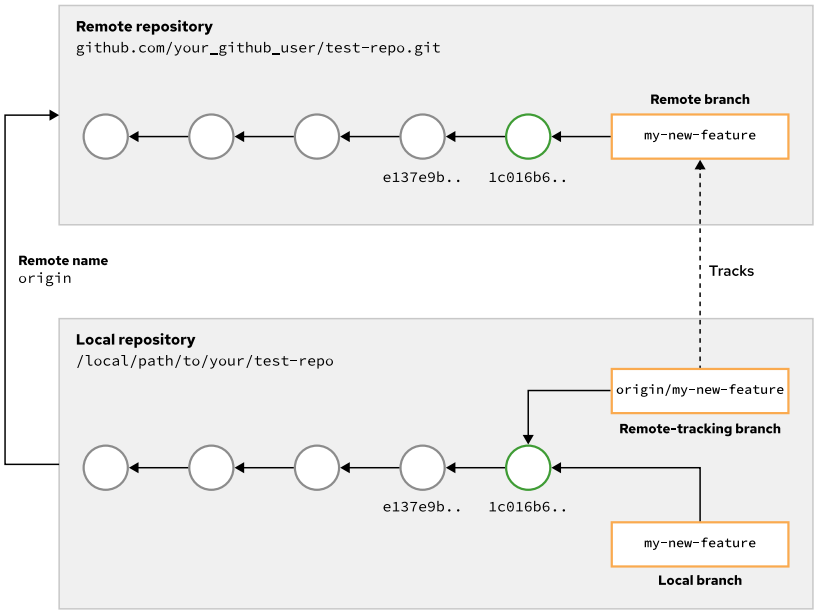


Figure 2.9: The remote branch is up to date with the local branch

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.

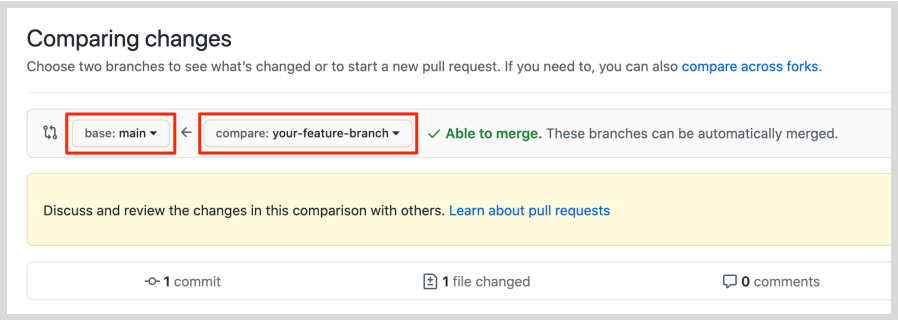


Figure 2.10: GitHub view to compare changes

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.

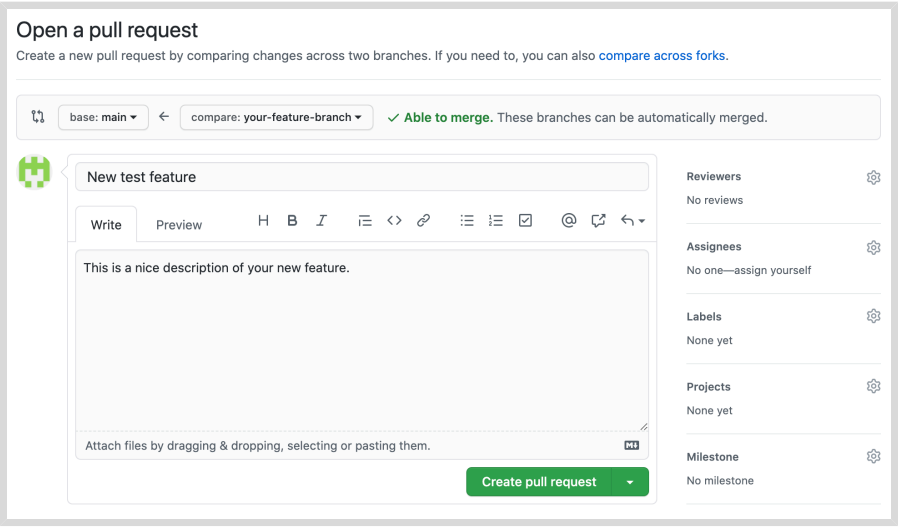


Figure 2.11: GitHub pull request form

> **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.

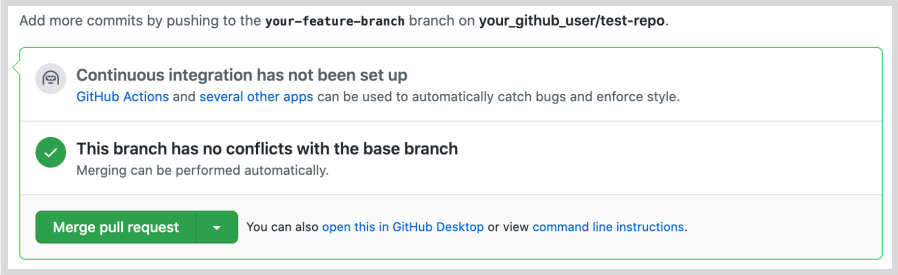


Figure 2.12: GitHub pull request merge pane

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.

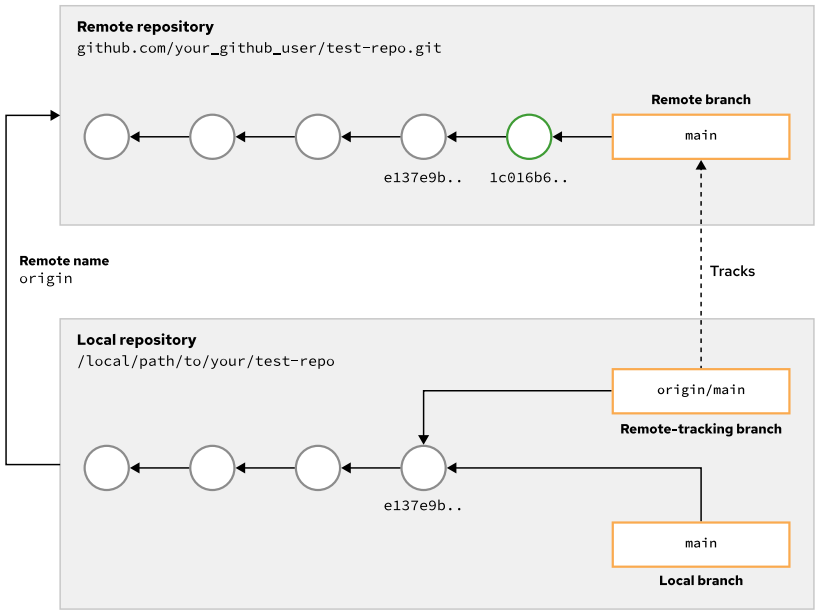


Figure 2.13: The local main and origin/main branches are behind the remote main branch

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.

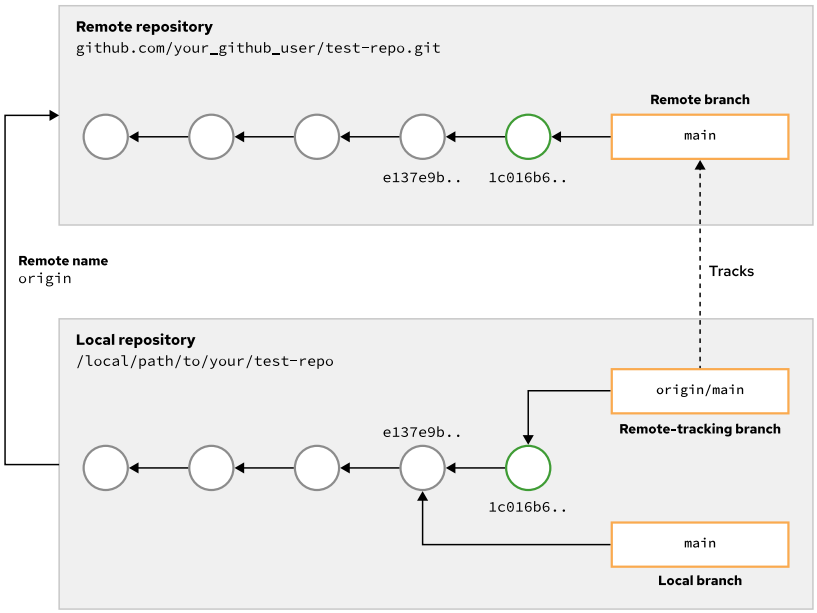


Figure 2.14: The local main branch is behind the remote-tracking origin/main branch

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...
```

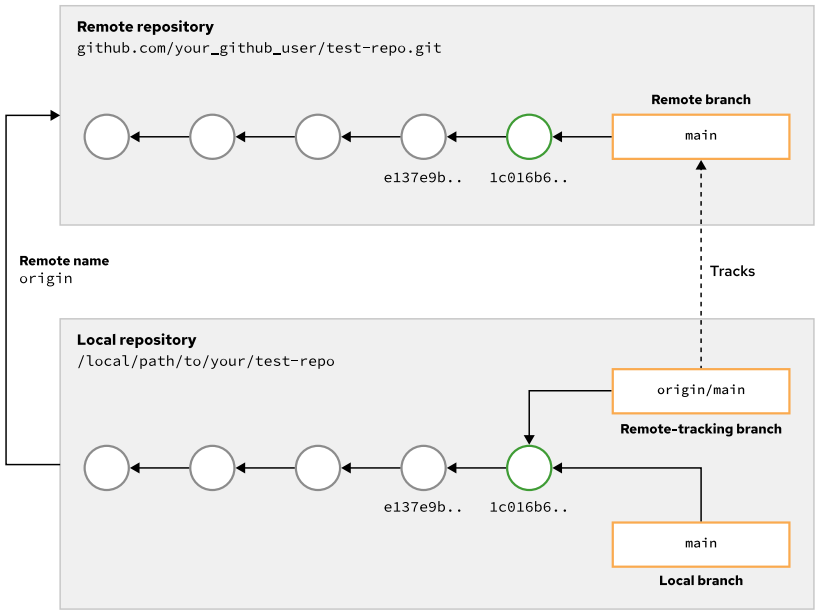


Figure 2.15: The local main branch is up to date with the remote main branch

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 (1)
Already on 'main'
Your branch is up to date with 'origin/main'.
[user@host your-local-repo]$ git pull origin main (2)
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 backlash 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 `~/DO400`. 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 DO400]$ mkdir hello-remote
```

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

```
[user@host DO400]$ cd hello-remote
[user@host hello-remote]$ git init .
Initialized empty Git repository in ...output omitted.../DO400/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`:

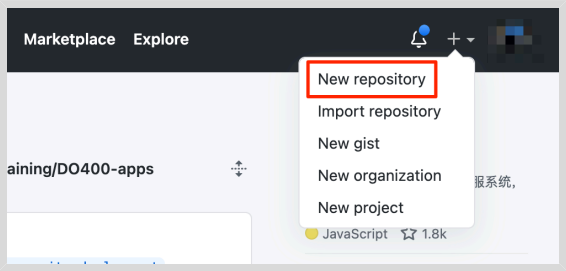
```
[user@host hello-remote]$ git branch -M main
```

2. Create a new repository in GitHub.

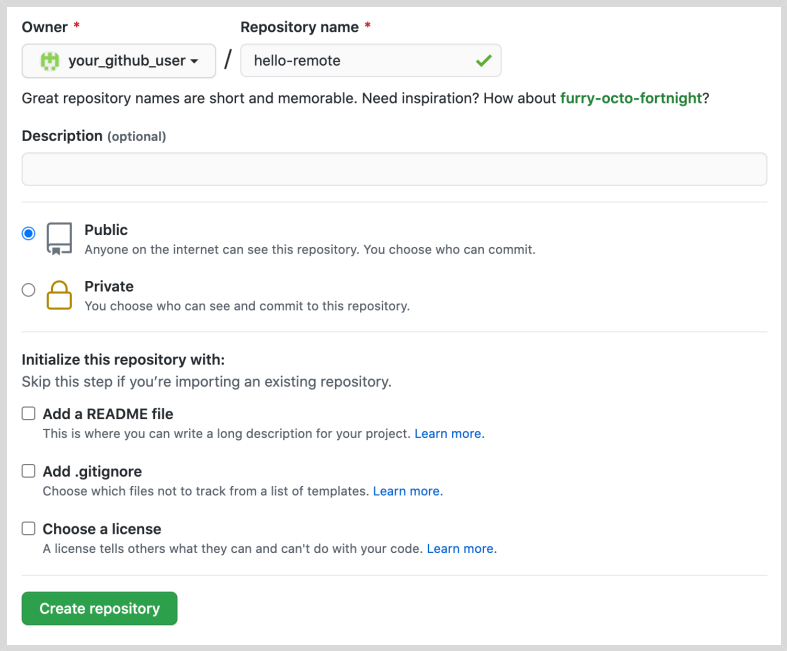
2.1. Open a web browser and navigate to `https://github.com`.

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**.



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

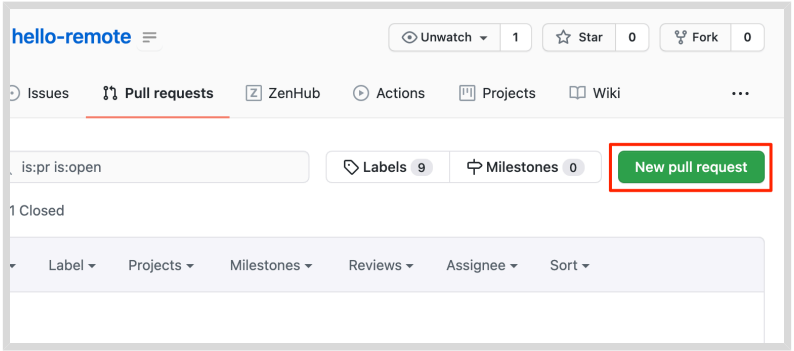
Use the `--set-upstream` parameter when pushing a new local branch to the remote for the first time.

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.

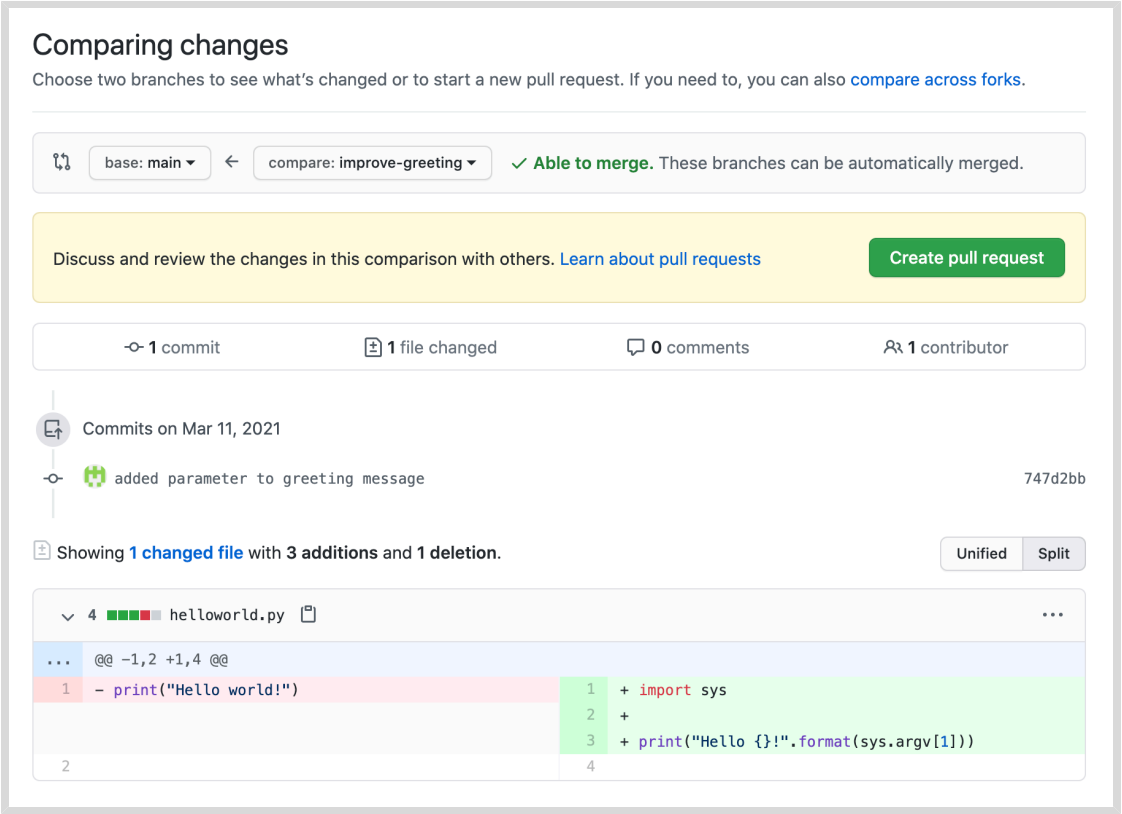
5. Review the changes introduced in the feature branch by using a pull request. After reviewing the code, merge the pull request to integrate the new feature into the `main` branch.

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.



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

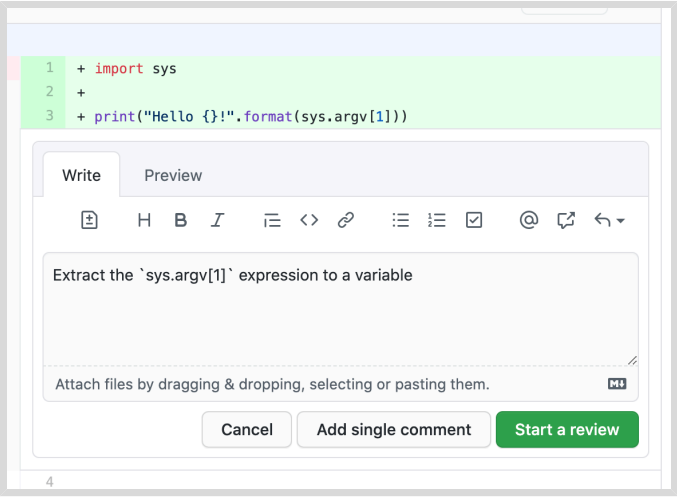
5.4. Click **Assignees** to assign the pull request to yourself. Click **Reviewers** to request reviews from other people.

Note

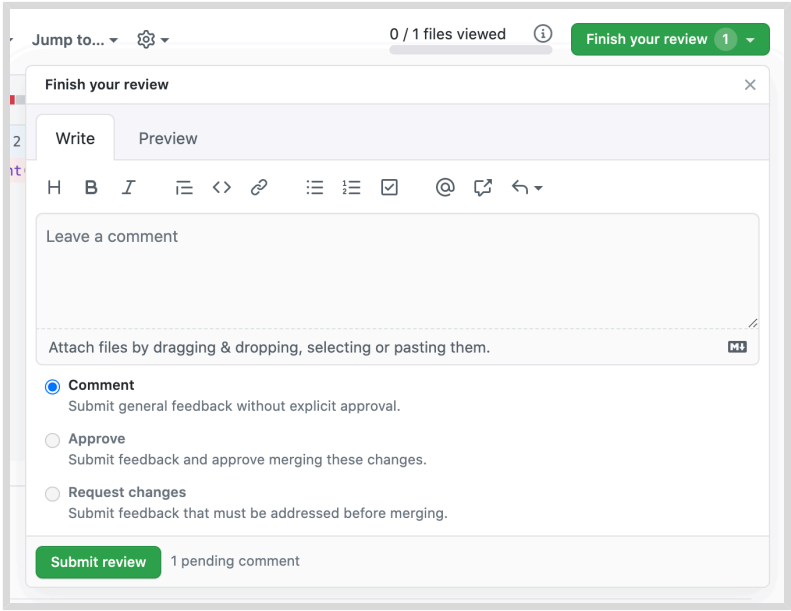
Normally, you would assign the pull request to someone else for peer review.

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`.

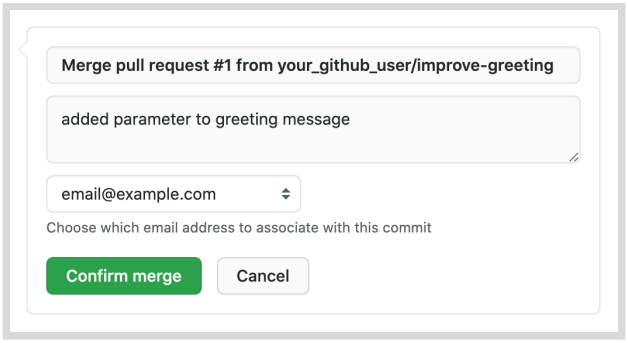
7. Merge the pull request.

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.

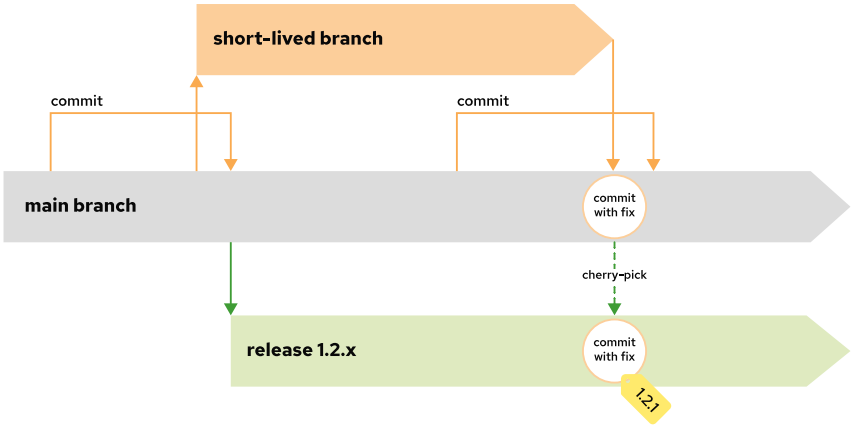


Figure 2.23: Branching structure in Trunk-based Development with a release branch

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.

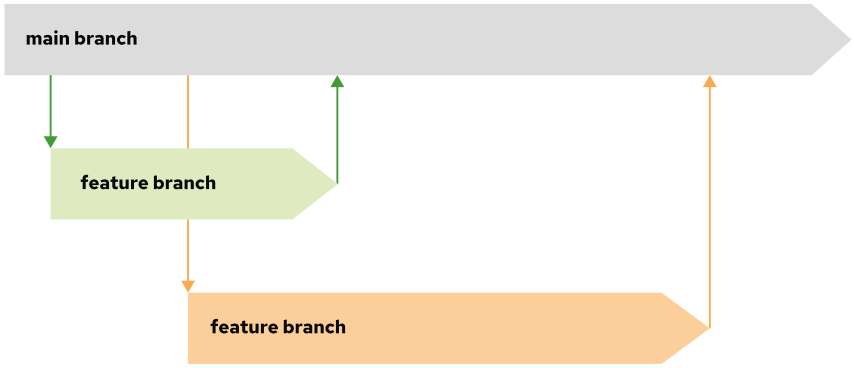


Figure 2.24: Branching structure in Feature Branching

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.

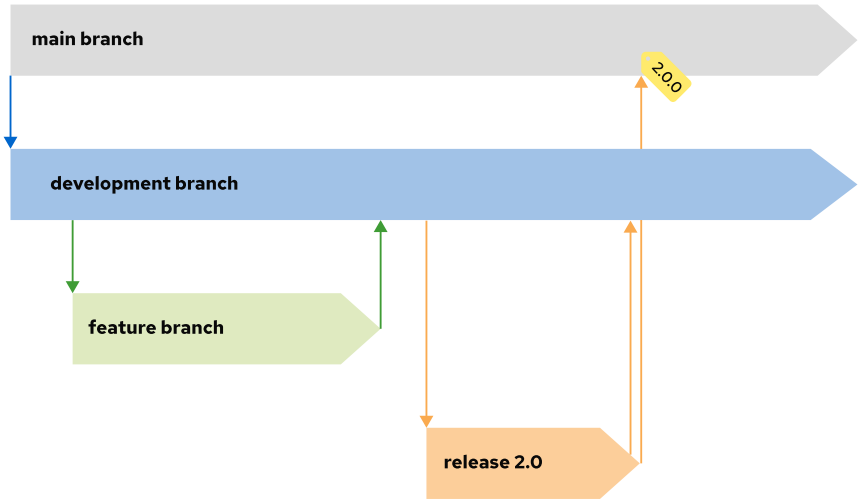


Figure 2.25: Branching structure in Git Flow

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.

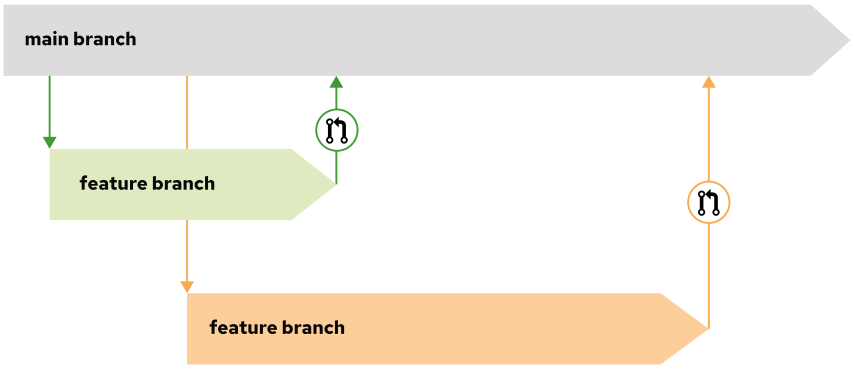


Figure 2.26: Branching structure in GitHub Flow

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.

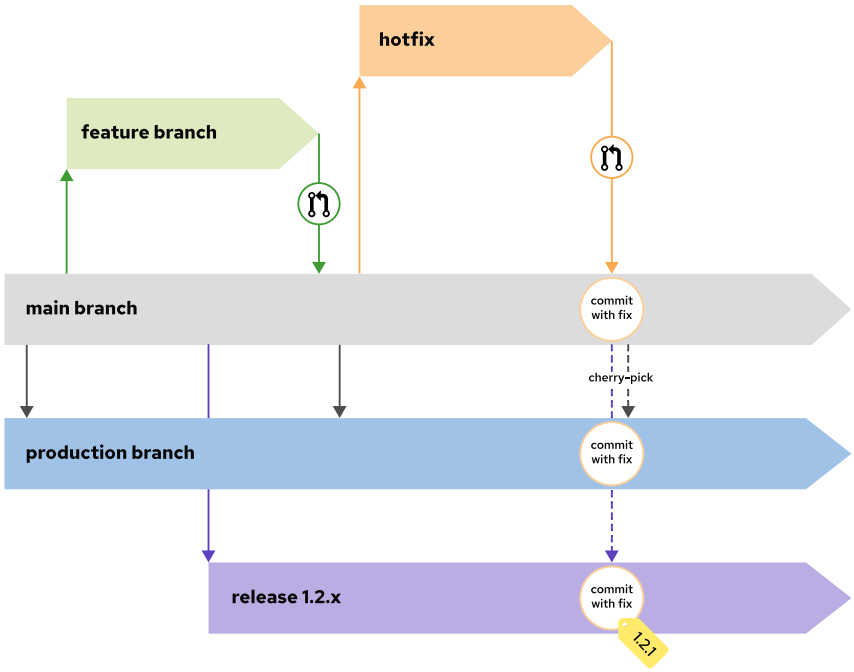


Figure 2.27: Branching structure in GitLab Flow

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/DO400-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 backlash 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 `~/DO400`. Start every guided exercise from this folder.

Instructions

1. Create your own GitHub organization. This organization will host your simulated third-party project.

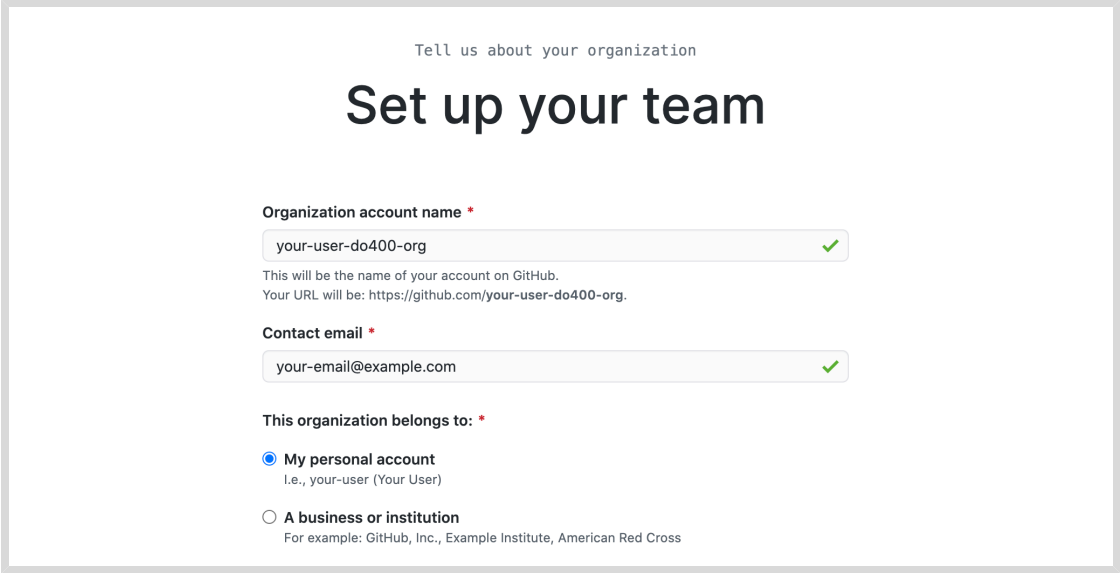
   1.1. Open a web browser and navigate to `https://github.com`.

   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**.

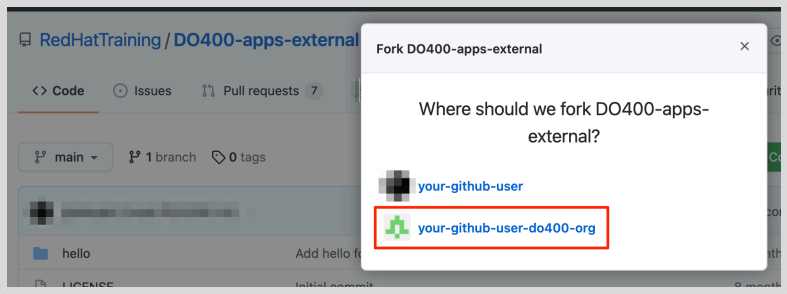


1.6. In the confirmation screen, click **Complete setup**. You must provide your password to complete this step.

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`.

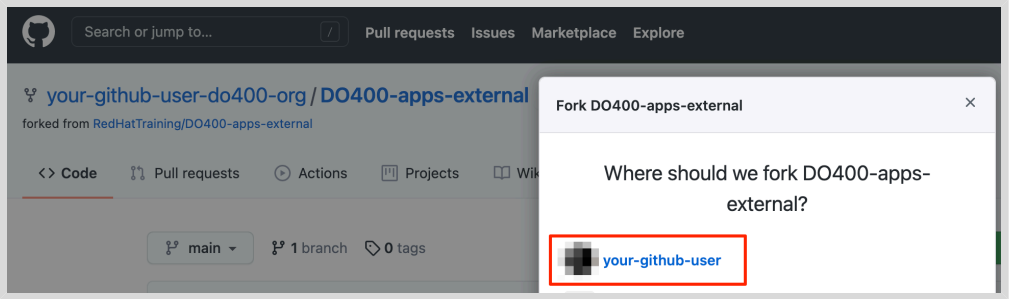
2.2. Click `Fork` in the upper right and select the *YOUR_GITHUB_USER*`-do400-org` organization as the namespace. This creates the *YOUR_GITHUB_USER*`-do400-org/DO400-apps-external` repository.



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 DO400]$ cd DO400-apps-external
[user@host DO400-apps-external]$ git tag 1.0.0
```

4.2. Use the `git tag` command to verify that the tag has been created:

```
[user@host DO400-apps-external]$ git tag
1.0.0
```

Note

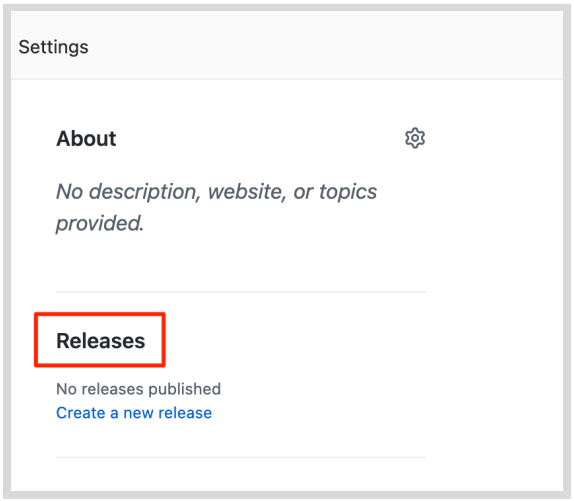
If Git shows the output by using a pager, press `q` to quit the pager.

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 DO400-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/DO400-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/DO400-apps-external`. Click the **Releases** link in the right pane.



4.5. On the releases page, click **Draft a new release**.

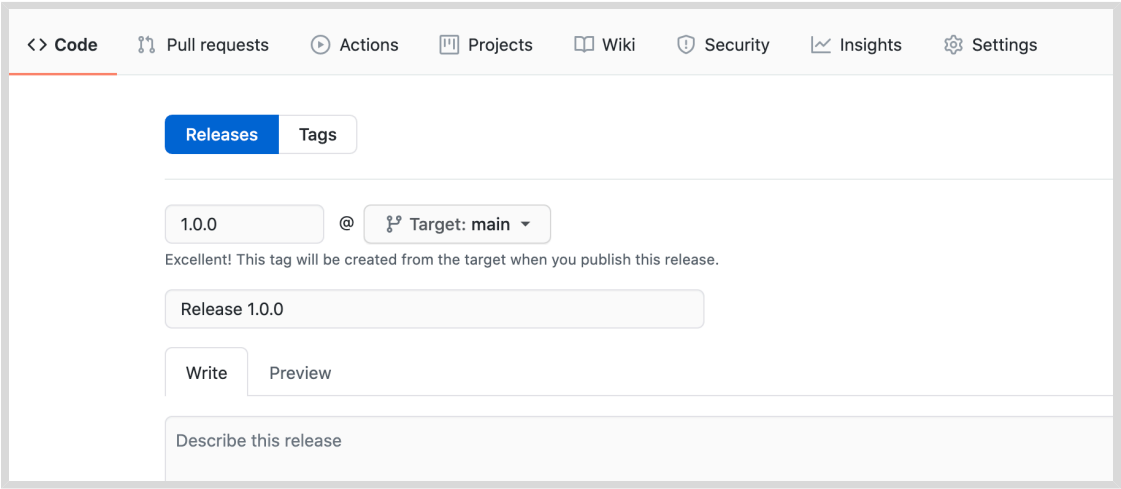
Note

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 DO400-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:

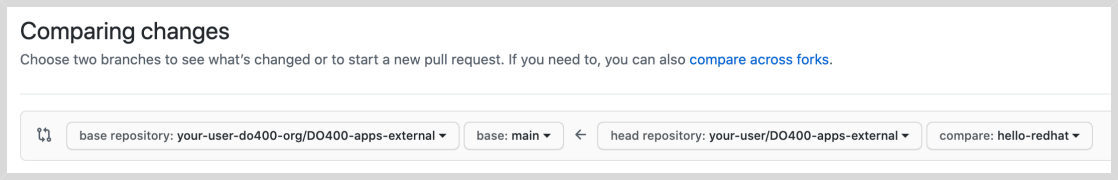
```
[user@host DO400-apps-external]$ git add hello
[user@host DO400-apps-external]$ git commit -m "added hello Red Hat"
[hello-redhat 60db2e8] added hello Red Hat
 1 file changed, 1 insertion(+)
 create mode 100644 hello/hellorh.py
```

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 DO400-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/DO400-apps-external/pull/new/
hello-redhat
remote:
To https://github.com/your_github_user/DO400-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/DO400-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`*`/DO400-apps-external`) into the `main` branch of the organization upstream repository (*`YOUR_GITHUB_USER`*`-do400-org/DO400-apps-external`). Next, click the **Create pull request** button to open the pull request creation form.



Important

Make sure you do not select `RedHatTraining/DO400-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 DO400-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*`/DO400-apps-external`). Enter your GitHub username and password if prompted.

```
[user@host DO400-apps-external]$ git pull origin main
...output omitted...
From https://github.com/your_github_user/DO400-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 DO400-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/DO400-apps-external`) and not the username fork (*YOUR_GITHUB_USER*`/DO400-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/DO400-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 DO400-apps-external]$ git remote add upstream \
https://github.com/YOUR_GITHUB_USER-do400-org/DO400-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 DO400-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/DO400-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 DO400-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 DO400-apps-external]$ git push origin main
...output omitted...
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/your_github_user/DO400-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/DO400-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 DO400-apps-external]$ git pull --tags
...output omitted...
From https://github.com/your_github_user/DO400-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 DO400-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 `DO400-apps-external` folder

```
[user@host DO400-apps-external]$ cd ..
[user@host DO400]$ rm -rf DO400-apps-external
```

Important

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

```
PS C:\Users\user\DO400> rm -Recur -Force DO400-apps-external
```

10.2. Remove the downstream and upstream repositories.

Go to your GitHub repository settings page at `https://github.com/YOUR_GITHUB_USER/DO400-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/DO400-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 backlash 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 DO400 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 DO400 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 DO400 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 DO400 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 backlash 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 DO400]$ git config --get user.name
[user@host DO400]$ git config --get user.email
```

   1.2. Use the `git config` command to configure your name.

```
[user@host DO400]$ git config --global user.name "YOUR USER NAME"
```

1.3. Use the `git config` command to configure your user email.

```
[user@host DO400]$ git config --global user.email YOUR@USER.EMAIL
```

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

```
[user@host DO400]$ git config --get user.name
Your User Name
[user@host DO400]$ 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 DO400 course.
```

2.1. Create a new directory to house the repository and navigate to it.

```
[user@host DO400]$ mkdir do400-git-lab
[user@host DO400]$ 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 DO400 course.
> EOF
```

Important

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

```
PS C:\Users\user\DO400\do400-git-lab> @"
>> # do400-git-lab
>> This is an example project repository for the DO400 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 DO400 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.

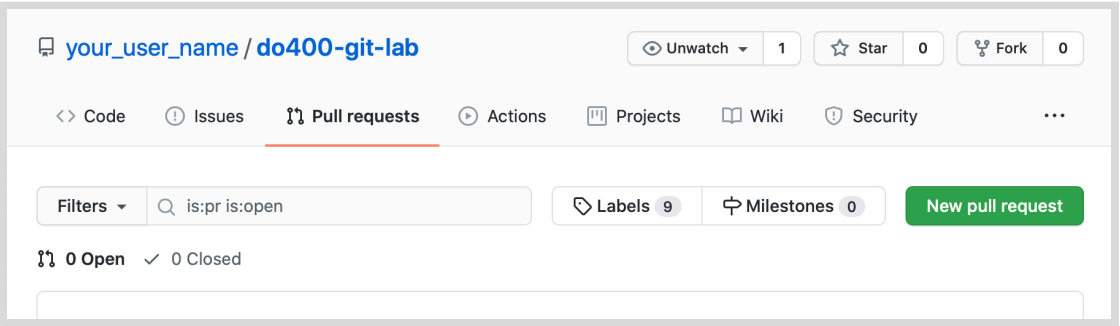
```
[user@host do400-git-lab]$ git add README.md
[user@host do400-git-lab]$ git commit -m 'added test repository notice'
[readme-update d4009df] added test repository notice
 1 file changed, 1 insertion(+)
```

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 DO400 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.

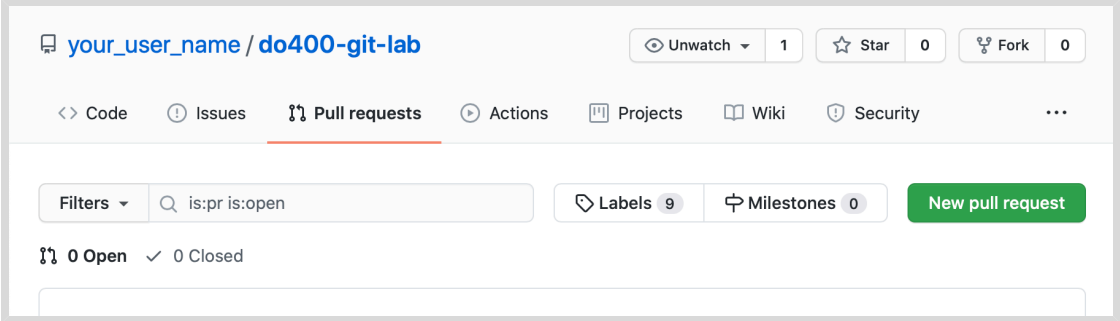
```
[user@host do400-git-lab]$ git add README.md
[user@host do400-git-lab]$ git commit -m 'added example repository notice'
[readme-example 0c67dea] added example repository notice
1 file changed, 1 insertion(+)
```

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 DO400 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 DO400 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.

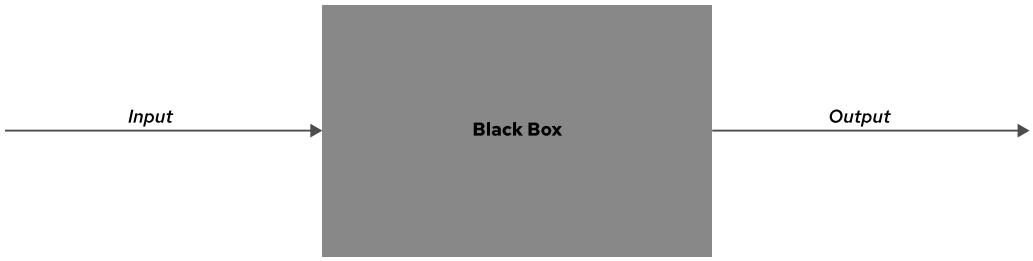


Figure 3.1: Black-box testing

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.

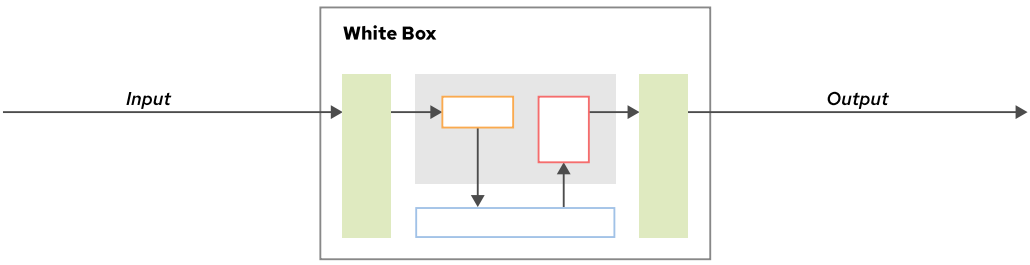


Figure 3.2: White-box testing

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.

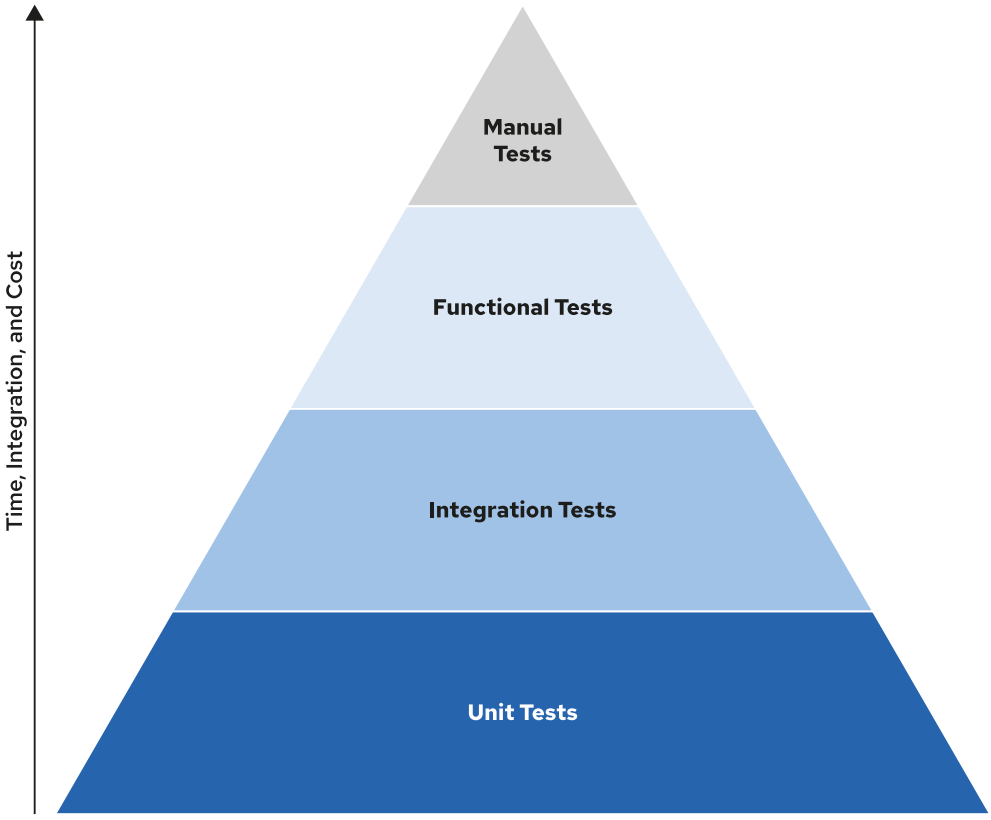


Figure 3.3: The testing pyramid

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.

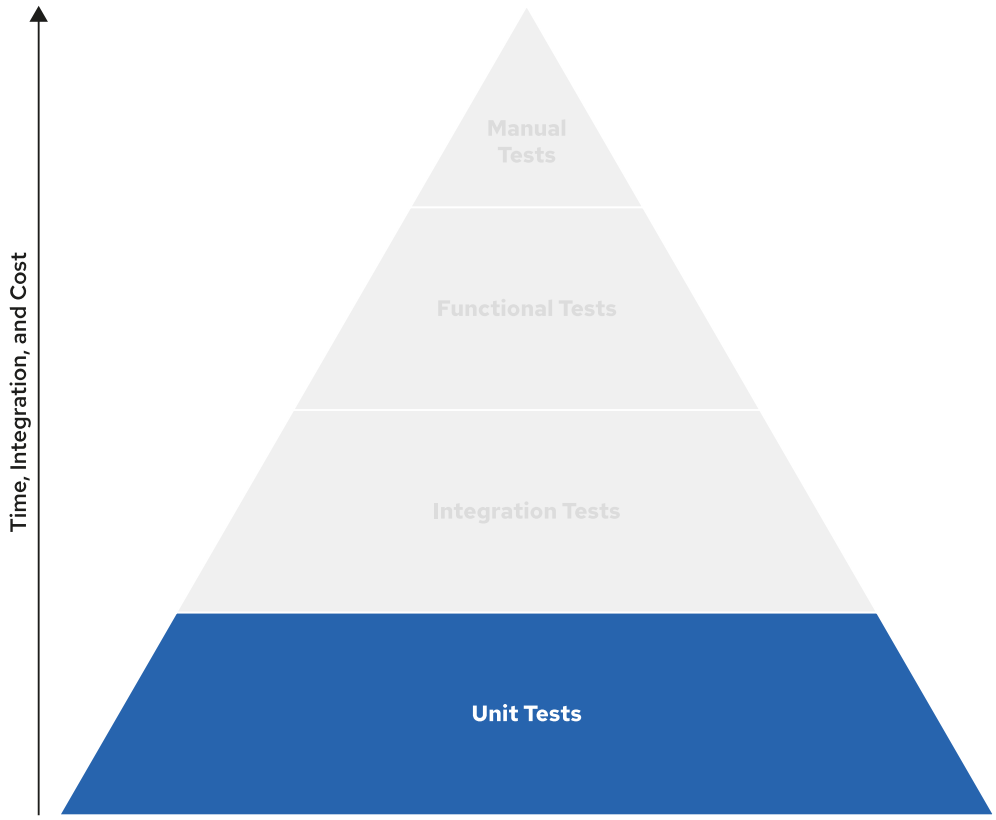


Figure 3.4: Unit Tests are the Base of the Test Pyramid

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() { (1)
    val fibonacciCalculator = new FibonacciCalculator() (2)
    val result = fibonacciCalculator.calculate(14) (3)
    assertEquals(377, result) (4)
}
```

(1) Test shows as functions invoked by the testing framework

(2) The **Given** section declares the instance of the `FibonacciCalculator` class needed to hold the `calculate` unit method.

(3) The **When** phase invokes the `calculate` method and retrieves the results.

(4) 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.

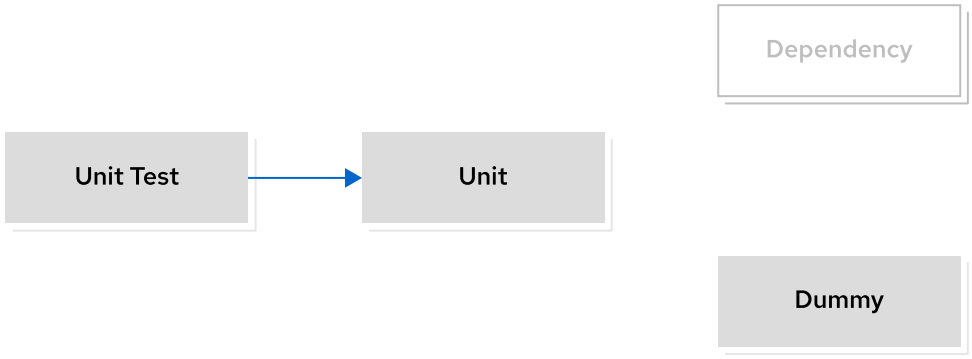


Figure 3.5: Dummy type example

Fakes

`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.

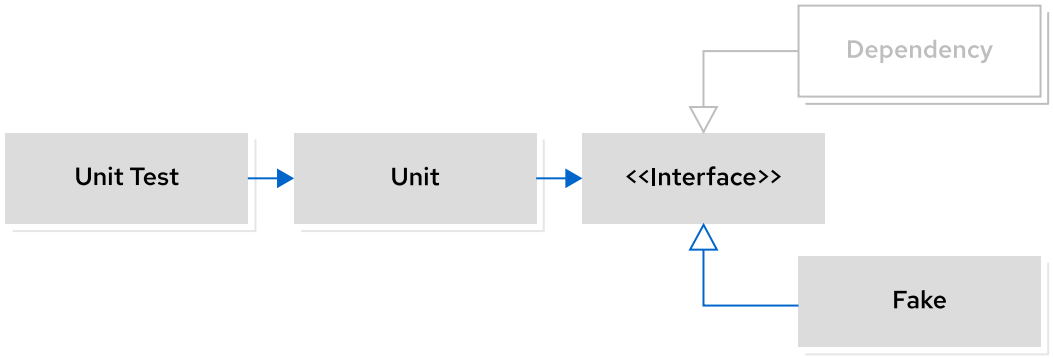


Figure 3.6: Fake type example

Stubs

`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.

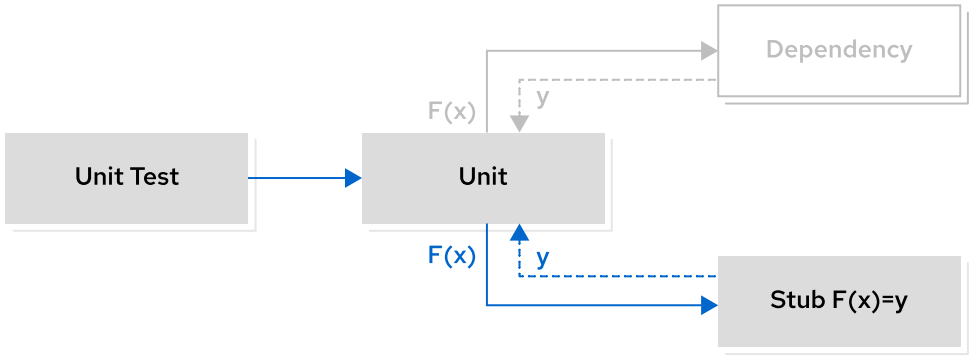


Figure 3.7: Stub type example

Mocks

`Mocks` are dependencies pre-programed 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.

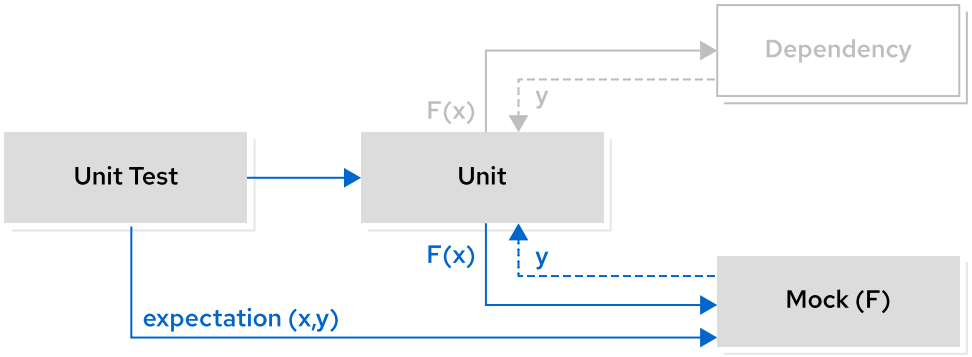


Figure 3.8: Mock type example

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.

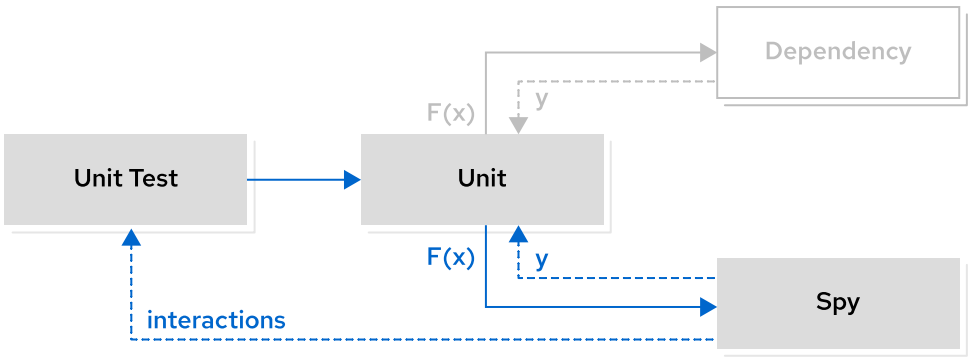


Figure 3.9: Spy type example

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 `DO400-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 `DO400-apps` fork cloned in your workstation.

Important

Try to keep your course work organized by creating a workspace folder, such as `~/DO400`. 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 `DO400-apps/school-library` application folder and checkout the `main` branch of the `DO400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host DO400]$ cd DO400-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] ------------------------------------------------------------------------
[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.

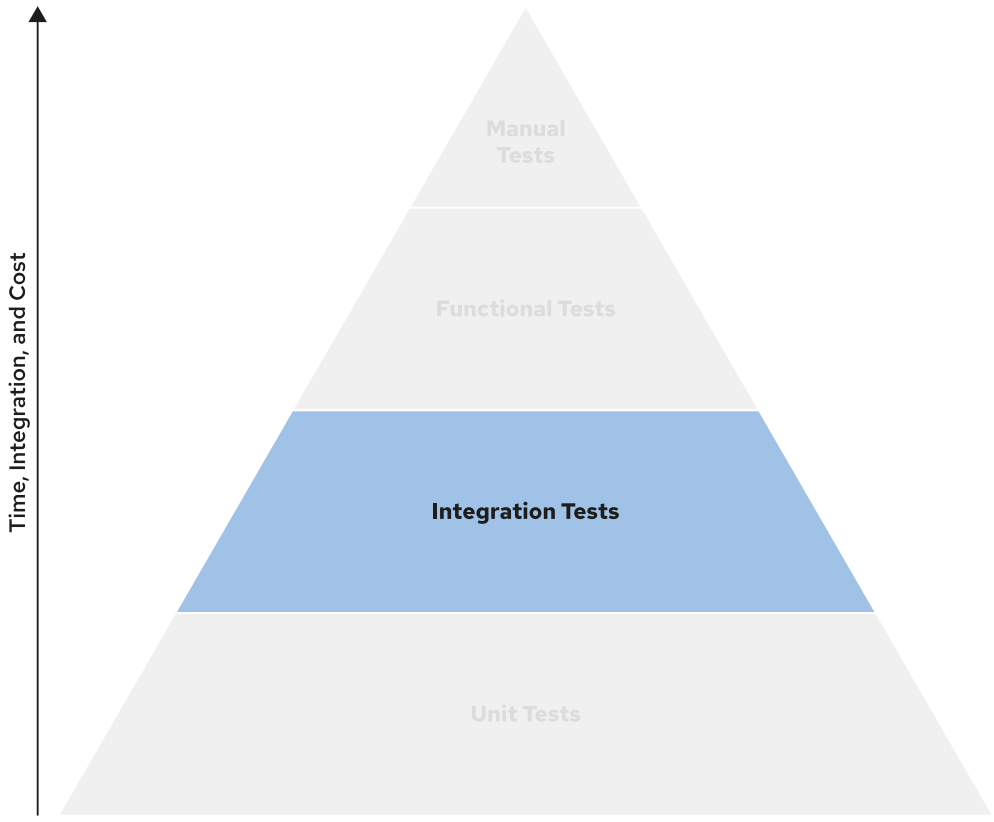


Figure 3.10: Integration Tests as the second level of the Test Pyramid

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(); (1)
}

@AfterEach
public void teardown() {
     tearDownTestDatabase(); (2)
}

@Test
public void testRepositoryCanReadFromDB() {
    // Given
    val repository = new ProductRepository(); (3)

    // When
    val productList = repository.getAll(); (4)

    // Then
    assertThat(productList, is(not(empty()))); (5)
}
```

The structure of this test is:

1. Starts the database and fills it with test data before each test.
2. Stops the database after each test.
3. Sets up the test prerequisites.
4. Calls the method to test.
5. 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 `DO400-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 `DO400-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 `~/DO400`. 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 `DO400-apps/shopping-cart` application folder and checkout the `main` branch of the `DO400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host DO400]$ cd DO400-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.

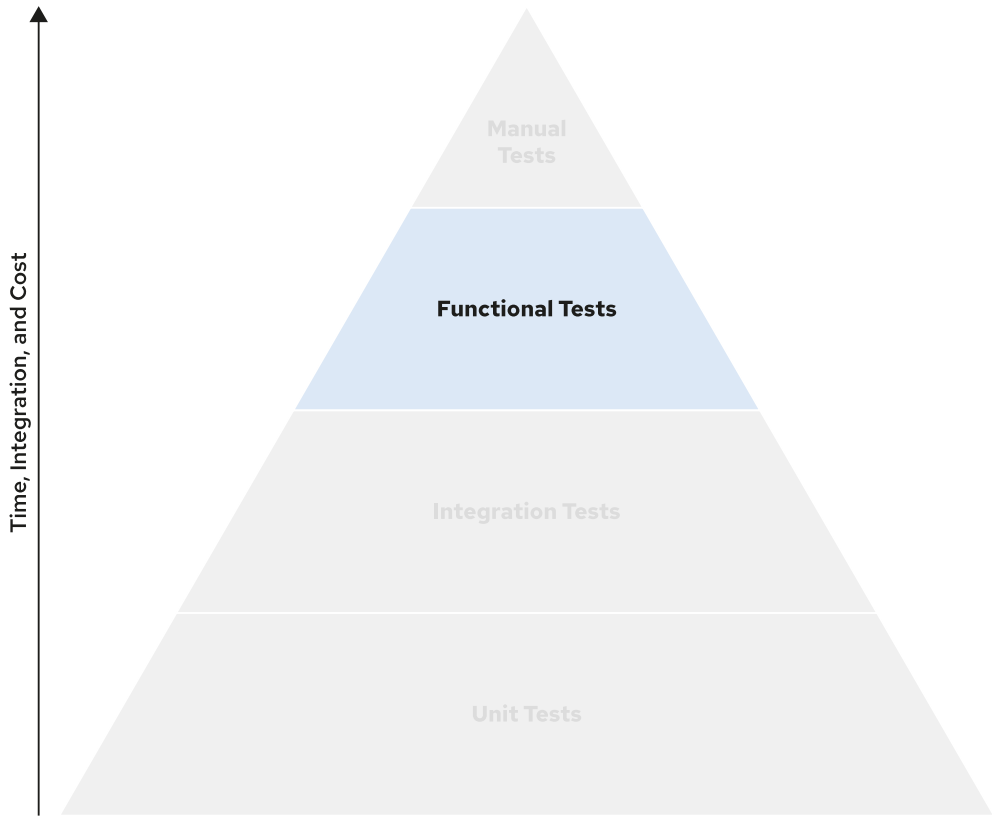
Building Functional Tests

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, espeically 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.

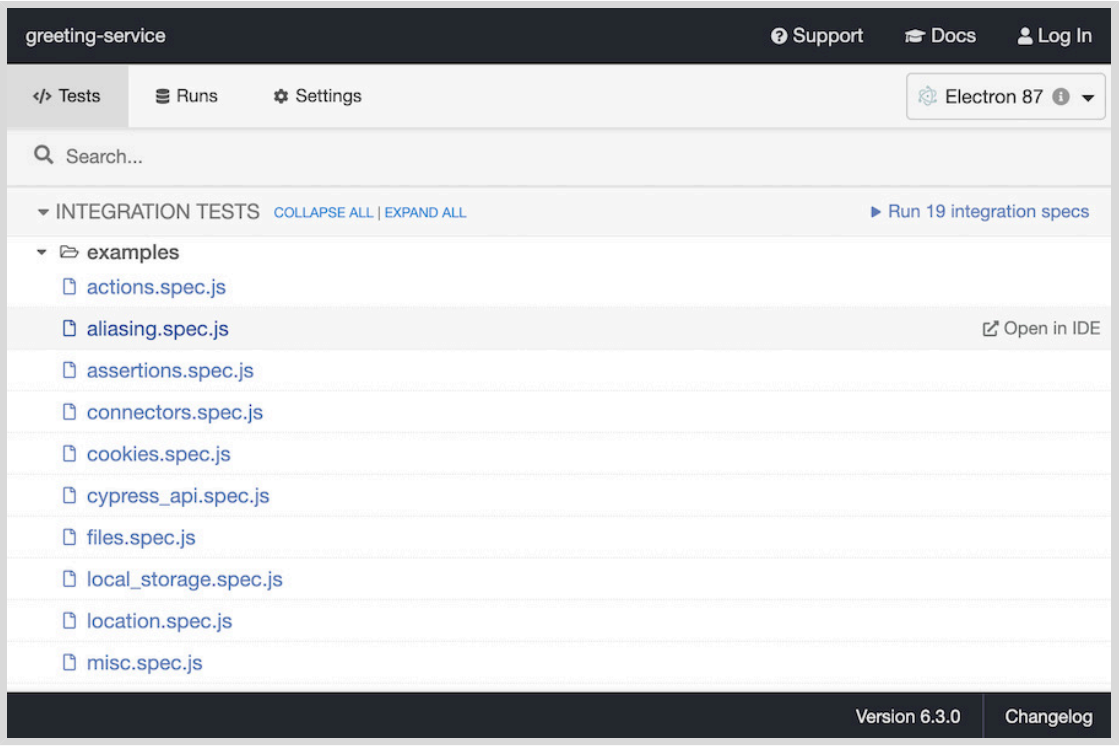


Figure 3.12: Cypress testing dialog

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`.

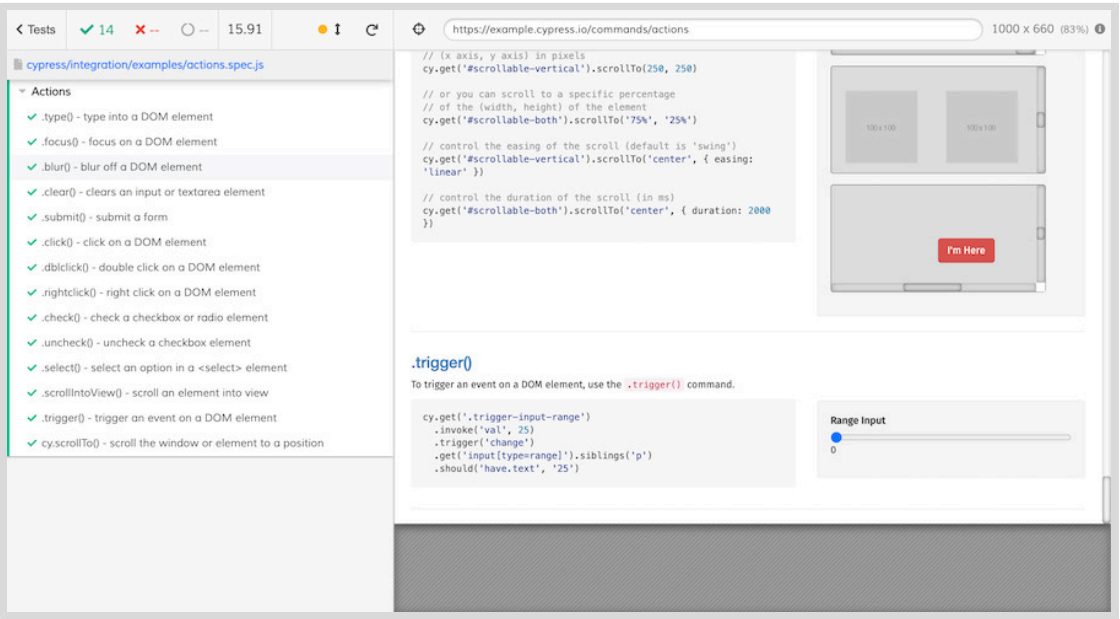


Figure 3.13: Example Cypress test execution

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 standardardize 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 `DO400-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 `DO400-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 `~/DO400`. 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 `DO400-apps/scoreboard` application folder and checkout the `main` branch of the `DO400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host DO400]$ cd DO400-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 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 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 `DO400-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/DO400-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 `~/DO400`. 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 DO400]$ cd DO400-apps
[user@host DO400-apps]$ git remote -v
origin https://github.com/your_github_user/DO400-apps.git (fetch)
origin https://github.com/your_github_user/DO400-apps.git (push)
[user@host DO400-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 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 `DO400-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/DO400-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 `~/DO400`. 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 DO400]$ cd DO400-apps
[user@host DO400-apps]$ git remote -v
origin https://github.com/your_github_user/DO400-apps.git (fetch)
origin https://github.com/your_github_user/DO400-apps.git (push)
[user@host DO400-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

...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

...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
- Introducing Test-driven Development (GE)
- Developing with TDD and Other Best Practices
- Developing with TDD and Other Best Practices (GE)
- Analyzing Code Quality
- Analyzing Code Quality (GE)
- Building Applications with Test-driven Development (Lab)
- Summary
- Executing Automated Tests in Pipelines
- Executing Automated Tests in Pipelines (GE)

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.

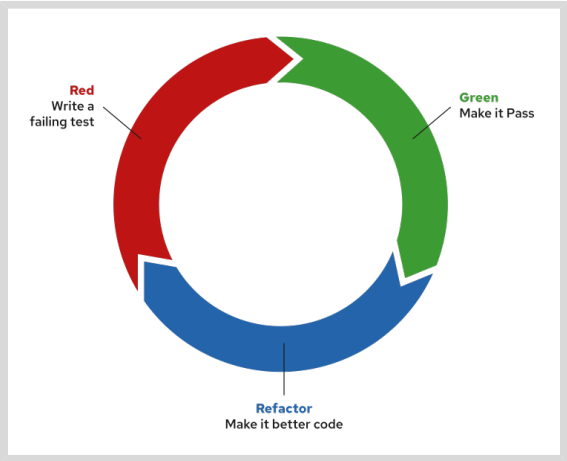


Figure 4.1: Red - Green - Refactor cycle

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/DO400-apps` under the `shipping-calculator` directory.

You will find the solution files for this exercise in the `solutions` branch of the `DO400-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 `DO400-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 `~/DO400`. 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 `DO400-apps/shipping-calculator` application folder and checkout the `main` branch of the `DO400-apps` repository to ensure you start this exercise from a known clean state.

```
[user@host DO400]$ cd DO400-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 most 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()
```

a. Start a new TDD cycle to develop more functionality. No refactor needed.
b. Join the two tests to unify the responsibilities of each of them into a single test that covers both scenarios.
c. 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.
d. 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
```

   a. The `HomeAutomation` class is decoupled from the lighting system and easily adapts to other lighting system vendors.
   b. The `HomeAutomation` class is coupled to a particular lighting system and does not easily adapt to other lighting system vendors.
   c. The code structure facilitates the task of testing the `HomeAutomation` class in isolation.
   d. The `AcmeLightSystem` dependency makes unit tests potentially slower and more difficult to setup.
   e. 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()
```

a. Start a new TDD cycle to develop more functionality. No refactor needed.

b. Join the two tests to unify the responsibilities of each of them into a single test that covers both scenarios.

c. 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.

d. 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
```

a. The `HomeAutomation` class is decoupled from the lighting system and easily adapts to other lighting system vendors.

b. The `HomeAutomation` class is coupled to a particular lighting system and does not easily adapt to other lighting system vendors.

c. The code structure facilitates the task of testing the `HomeAutomation` class in isolation.

d. The `AcmeLightSystem` dependency makes unit tests potentially slower and more difficult to setup.

e. 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 asses 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> (1)
  <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> (2)
      </goals>
    </execution>
  </executions>
</plugin>
```

(1) 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:

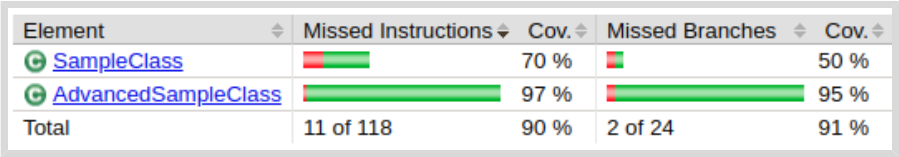
```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.5</version>
  <executions>
    <execution>
      <id>default-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>default-report</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 = Edges - 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.

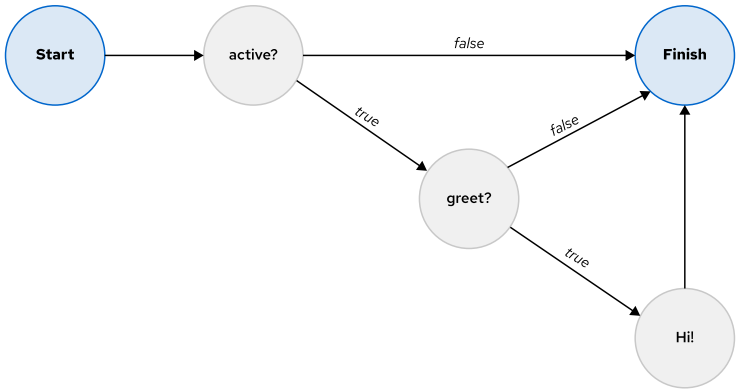


Figure 4.3: Execution flow graph

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 you 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> (1)
  <version>3.13.0</version>
  <configuration>
    <minimumTokens>10</minimumTokens>
    <printFailingErrors>true</printFailingErrors>
    <linkXRef>false</linkXRef>
    <rulesets>
      <ruleset>${pom.basedir}/pmd-rulesets.xml</ruleset> (2)
    </rulesets>
  </configuration>
</plugin>
```

(1) The Maven PMD plug-in

(2) 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 tools 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 `DO400-apps` repository, within the `simple-calculator` folder.

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 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 `DO400-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 `VSCodium`, which supports syntax highlighting for editing source files.

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. Get the source code of the sample application.

    1.1. From your workspace folder, navigate to the `DO400-apps` application folder and checkout the `main` branch to ensure you start this exercise from a known clean state.

```
[user@host DO400]$ cd DO400-apps
[user@host DO400-apps]$ git checkout main
...output omitted...
```

    1.2. Copy the contents of `~/DO400/DO400-apps/simple-calculator` to the `~/DO400/quality-checks` directory. Navigate to the new directory.

```
[user@host DO400-apps]$ cp -Rv ~/DO400/DO400-apps/simple-calculator/ \
~/DO400/quality-checks
...output omitted...
[user@host DO400-apps]$ cd ~/DO400/quality-checks
[user@host quality-checks]$
```

Important

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

```
PS C:\Users\user\DO400\DO400-apps> Copy-Item -Path simple-calculator `
>> -Destination ~\DO400\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.

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**.

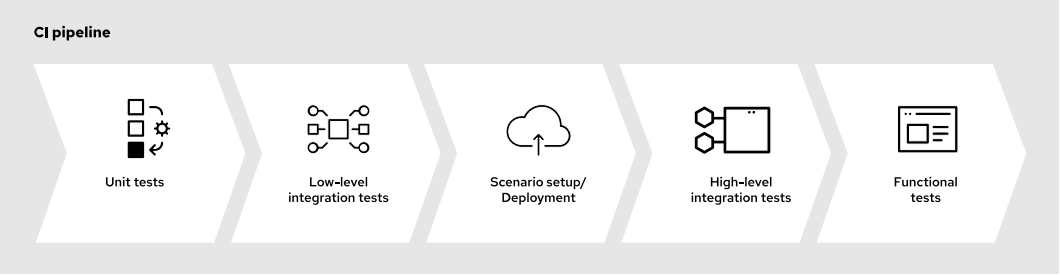


Figure 4.4: Tests order in a CI pipeline

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.

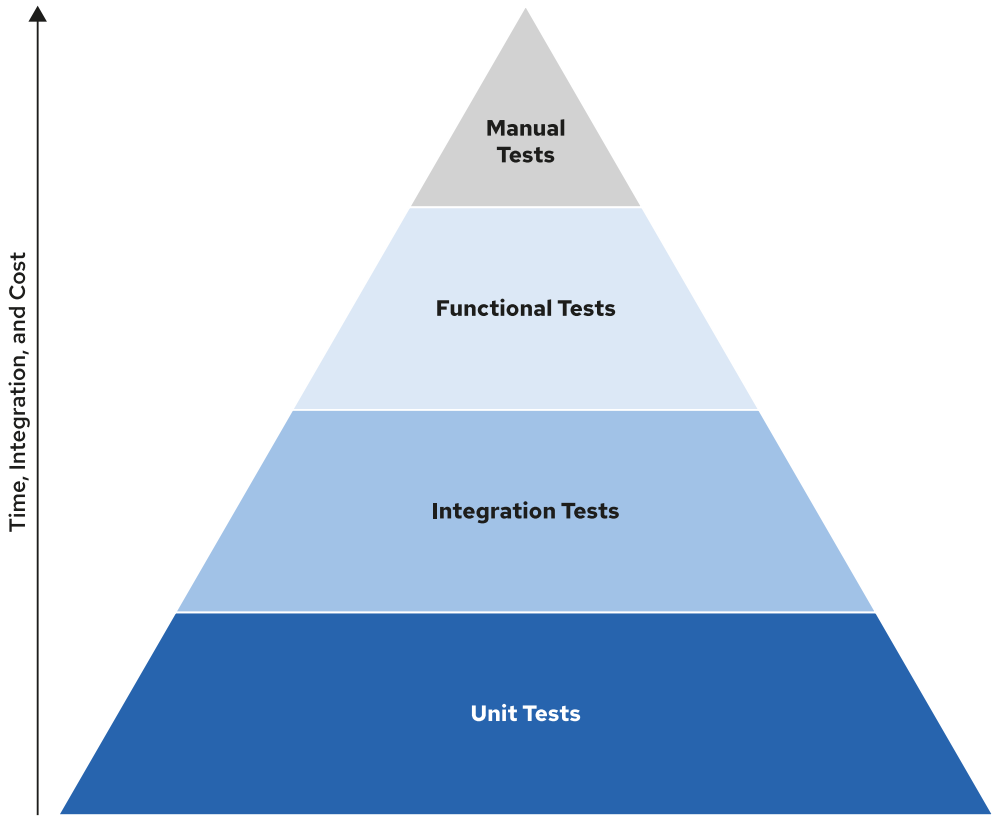


Figure 4.5: The testing pyramid

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 `DO400-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 `~/DO400`. 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 `DO400-apps` repository. Create a branch named `jenkins-testing`.

```
[user@host DO400]$ cd DO400-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 `DO400-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

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 `DO400-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/DO400-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 `~/DO400`. Start the guided exercise from this folder.

```
[user@host DO400]$ cd DO400-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 `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 `Substract` 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 `Substract.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.

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.

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.

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.

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 `DO400-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/DO400-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 `~/DO400`. Start the guided exercise from this folder.

```
[user@host DO400]$ cd DO400-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 `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 `Substract` 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 `Substract.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.Substract;
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(substract, 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>
  <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/Substract.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`, `Substract` 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`, `Substract`, 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`, `Substract` 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.Substract;

@ApplicationScoped
public class OperationRegistry {

    @Inject
    Add add;

    @Inject
    Substract substract;

    @Inject
    Multiply multiply;

    @Inject
    Identity identity;

    private List<Operation> operations;

    @PostConstruct
    protected void buildOperationList() {
        setOperations(List.of(substract, 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.