7

Continuous Integration and Continuous Delivery

One of the main pillars of **development-operations** (**DevOps**) culture is the implementation of **continuous integration** (**CI**) and deployment processes, as we explained in *Chapter 1*, *The DevOps Culture and Infrastructure as Code Practices*.

In the previous chapter, we looked at the use of Git with its command lines and usage workflow, and in this chapter, we will look at the important role Git has in the CI/CD workflow.

CI is a process that provides rapid feedback on the consistency and quality of code to all members of a team. It occurs when each user's code commit retrieves and merges the code from a remote repository, compiles it, and tests it.

Continuous delivery (**CD**) is the automation of the process that deploys an application in different stages (or environments).

In this chapter, we will learn the principles of the CI/CD process as well as its practical use with different tools such as **Jenkins**, **Azure Pipelines**, and **GitLab CI**. For each of these tools, we will present the advantages, disadvantages, and best practices, and look at a practical example of implementing a CI/CD pipeline.

You will learn about the concept of a CI/CD pipeline. After this, we will explore package managers and the role they play in the pipeline.

Then, you will learn how to install Jenkins and, finally, build a CI/CD pipeline on Jenkins, Azure Pipelines, and GitLab CI.

This chapter covers the following topics:

- CI/CD principles
- Using a package manager in the CI/CD process
- Using Jenkins for CI/CD implementation
- Using Azure Pipelines for CI/CD
- Using GitLab CI

Technical requirements

The only requirement for this chapter is to have Git installed on your system, as detailed in the previous chapter.

The source code for this chapter is available at `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP07`.

Check out the following video to see the code in action: `https://bit.ly/3s9G8Pj`.

CI/CD principles

To implement a CI/CD pipeline, it is important to know the different elements that will be required to build an efficient and safe pipeline. In order to understand the principles of CI/CD, the following diagram shows the different steps of a CI/CD workflow, which we already saw in *Chapter 1*, *The DevOps Culture and Infrastructure as Code Practices*:

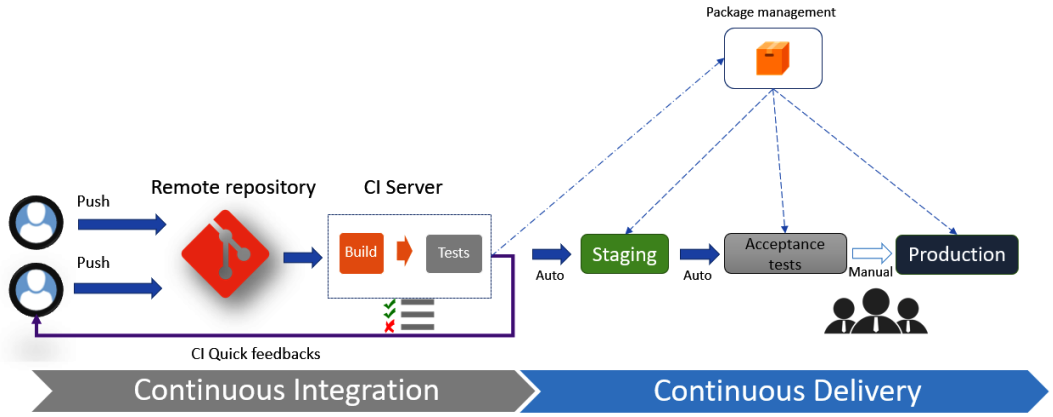

Figure 7.1 – CI/CD workflow

Let's look in detail at each of these steps in order to list the artifacts of the CI/CD process.

CI

The CI phase checks the code archived by the team members. It must be executed on each commit that has been pushed to the remote repository.

The setting up of a Git-type **source control version** (**SCV**) is a necessary prerequisite that makes it possible to centralize the code of all the members of a team.

The team will have to decide on a code branch that will be used for CI. For example, we can use the `master` branch, or the `develop` branch as part of GitFlow; it just needs to be an active branch that very regularly centralizes code changes.

In addition, CI is achieved by an automatic task suite that is executed on a server, following similar patterns executed on a developer's laptop that has the necessary tools for CI; this server is called the **CI server**.

The CI server can be either of the on-premises type, installed in the company data center, such as Jenkins or TeamCity, or perhaps a cloud type that we don't have to worry about installing and maintaining, such as Azure Pipelines or GitLab CI.

The tasks performed during the CI phase must be automated and take into account all the elements that are necessary for the verification of the code.

These tasks are generally the compilation of code and the execution of unit tests with code coverage. We can also add static code analysis with SonarQube (or SonarCloud), which we will look at in *Chapter 12*, *Static Code Analysis with SonarQube*.

At the end of the verification tasks, in many cases, the CI generates an application package that will be deployed on the different environments (also called **stages**).

To be able to host this package, we need a package manager, also called a **repository manager**, which can be on-premises (installed locally) such as Nexus, Artifactory, or ProGet, or a **software-as-a-service** (**SaaS**) solution such as Azure Pipelines, Azure Artifacts, or the GitHub Packages registry. This package must also be neutral in terms of environment configuration and *must* be versioned in order to deploy the application in a previous version if necessary.

CD

Once the application has been packaged and stored in a package manager during CI, the CD process is ready to retrieve the package and deploy it in different environments.

The deployment in each environment consists of a succession of automated tasks that are also executed on a remote server that has access to the different environments.

It is, therefore, necessary to involve Dev, Ops, and also the security team in the implementation of CI/CD tools and processes. It will, indeed, be this union of people with the tools and processes that will deploy applications on the different servers or cloud resources, respecting the network rules but also the company's security standards.

During the deployment phase, it is often necessary to modify the configuration of the application in the generated package in order for it to be adapted to the target environment. It is, therefore, necessary to integrate a **configuration manager** that is already present in common CI/CD tools such as Jenkins, Azure Pipelines, or Octopus Deploy. In addition, when there is a new configuration key, it is good practice for every environment, including production, to be entered with the involvement of the Ops team.

Finally, the triggering of a deployment can be done automatically, but for environments that are more critical (for example, production environments), heavily regulated companies may have gateways that require a manual trigger with checks on the people authorized to trigger the deployment.

The different tools for setting up a CI/CD pipeline are listed here:

- An SCV
- A package manager
- A CI server
- A configuration manager

But let's not forget that all these tools will only be really effective in delivering added value to the product if the Dev and Ops teams work together around them.

We have just looked at the principles of implementing a CI/CD pipeline. In the rest of the chapter, we will look at the practical implementation with different tools, starting with package managers.

Using a package manager in the CI/CD process

A package manager is a central repository to centralize and share packages, development libraries, tools, and software.

For consumer clients that use package managers, the benefit is the possibility to track, update, install, and remove installed packages.

There are many public package managers, such as NuGet, **Node Package Manager** (**npm**), Maven, Bower, and Chocolatey, that provide frameworks or tools for developers in different languages and platforms.

The following screenshot is from the NuGet package manager, which publicly provides more than 150,000 .NET frameworks:

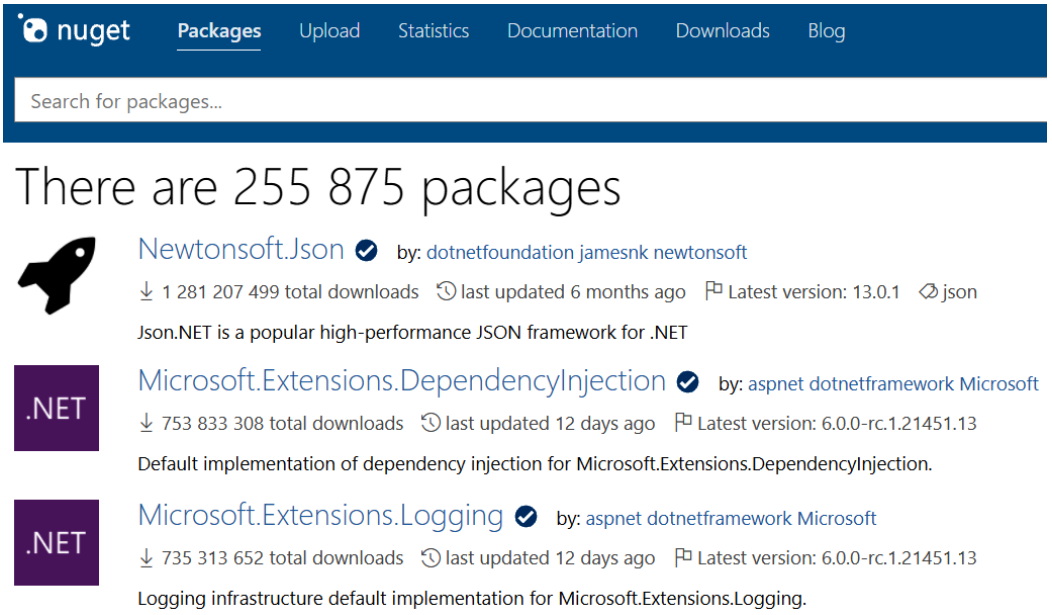

Figure 7.2 – NuGet package manager

Azure Artifacts has the advantage of being in SaaS offering mode, so there is no installation or infrastructure to manage; for more information, the documentation is available here: `https://azure.microsoft.com/en-us/services/devops/artifacts/`.

We finished this overview of package managers with the local NuGet server instance, NPM, Nexus, and Azure Artifacts. Of course, there are many other package manager tools that should be considered according to the company's needs.

After looking at package managers, we will now implement a CI/CD pipeline with a well-known tool called **Jenkins**.

Using Jenkins for CI/CD implementation

Jenkins is one of the oldest CI tools, initially released in 2011. It is open source and developed in Java.

Jenkins has become famous thanks to the large community working on it and its plugins. Indeed, there are more than 1,500 Jenkins plugins that allow you to perform all types of actions within your jobs. And if, despite everything, one of your tasks does not have a plugin, you can create one yourself.

In this section, we will look at the installation and configuration of Jenkins and will create a CI Jenkins job that will be executed during the commit of code that is in a Git repository.

The source code of the demonstration application is a Java project that is open source and available on the GitHub Microsoft repository space here: `https://github.com/microsoft/MyShuttle2`. To be able to use it, you need to fork it into your GitHub account.

Before talking about the Jenkins job, we will see how to install and configure Jenkins.

Installing and configuring Jenkins

Jenkins is a cross-platform tool that can be installed on any type of support, such as **virtual machines** (**VMs**) or even Docker containers. Its installation documentation is available here: `https://jenkins.io/doc/book/installing/`.

For our demo, and to quickly access the configuration of a CI/CD pipeline, we will use Jenkins on an Azure VM. In fact, Azure Marketplace contains a VM with Jenkins and its prerequisites already installed.

These steps show how to create an Azure VM with Jenkins and its basic configuration:

1. To get all the steps to create an Azure VM with Jenkins already installed, read the documentation available here: `https://docs.microsoft.com/en-us/azure/jenkins/install-jenkins-solution-template`.

   The following screenshot shows Jenkins integration on Azure Marketplace:

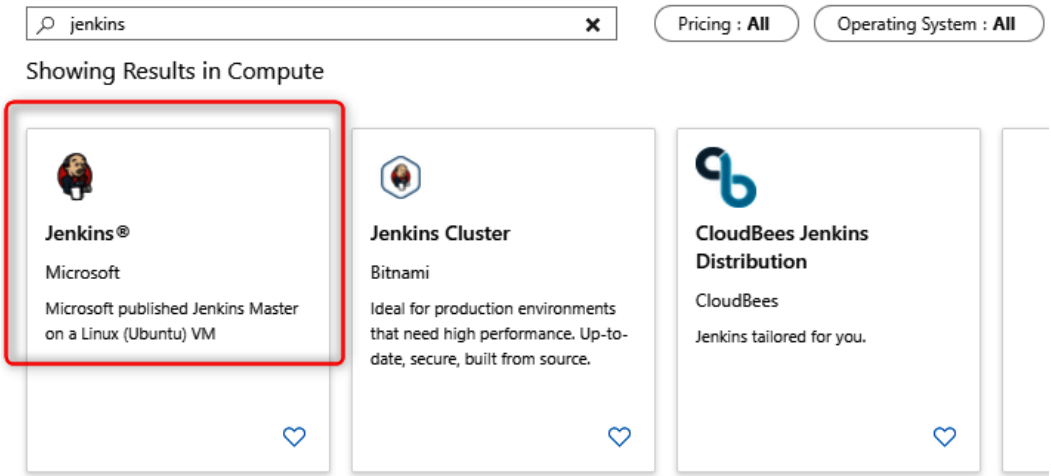

Figure 7.5 – Jenkins on Azure Marketplace

2. Once installed and created, we will access Jenkins in the browser by providing its **Uniform Resource Locator** (**URL**) in the Azure portal in the **DNS name** field, as shown in the following screenshot:

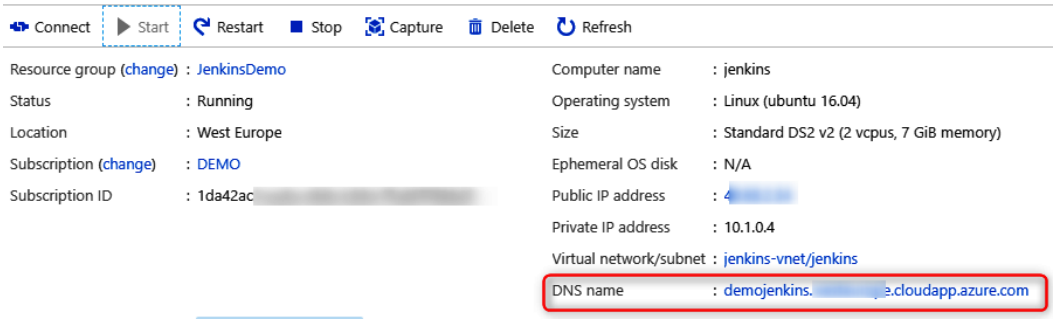

Figure 7.6 – Azure Domain Name System (DNS) name of Jenkins

3. Follow the displayed instructions on the Jenkins home page to enable access to this Jenkins instance via secure **Secure Sockets Layer** (**SSL**) tunneling. For more details about this step, read the documentation at `https://docs.microsoft.com/en-us/azure/jenkins/install-jenkins-solution-template#connect-to-jenkins` and this article: `https://jenkins.io/blog/2017/04/20/secure-jenkins-on-azure/`.
4. Then, follow the configuration instructions on the **Unlock Jenkins** message displayed on the Jenkins screen. Once the configuration is complete, we get Jenkins ready to create a CI job.

In order to use GitHub features in Jenkins, we also installed the **GitHub integration plugin** from the Jenkins plugin management by following this documentation: `https://jenkins.io/doc/book/managing/plugins/`.

The following screenshot shows the installation of the GitHub plugin:

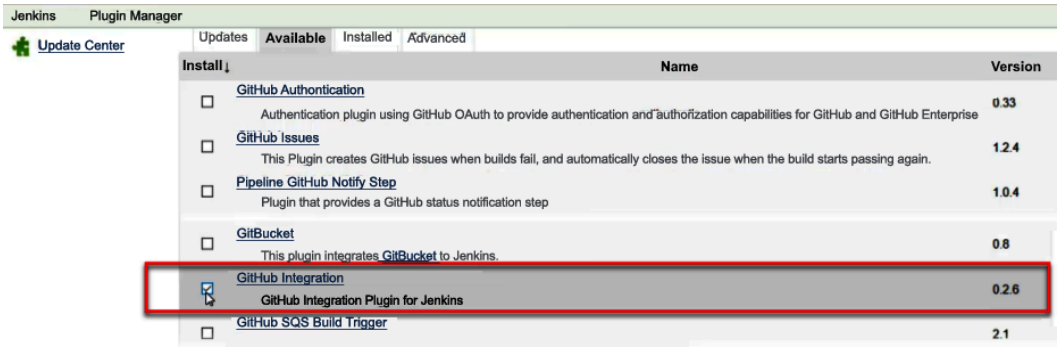

Figure 7.7 – Jenkins GitHub integration

Now that we have installed the GitHub plugin in Jenkins, let's look at how to configure GitHub with a webhook for its integration with Jenkins.

Configuring a GitHub webhook

In order for Jenkins to run a new job, we must first create a webhook in the GitHub repository. This webhook will be used to notify Jenkins as soon as a new push occurs in the repository.

To do this, follow these steps:

1. In the GitHub repository, go to the **Settings** | **Webhooks** menu.
2. Click on the **Add Webhook** button.

3. In the **Payload URL** field, fill in the URL address of Jenkins followed by `/github-webhook/`, leave the secret input as it is, and choose the **Just the push event** option.
4. Validate the webhook.

   The following screenshot shows the configuration of a GitHub webhook for Jenkins:

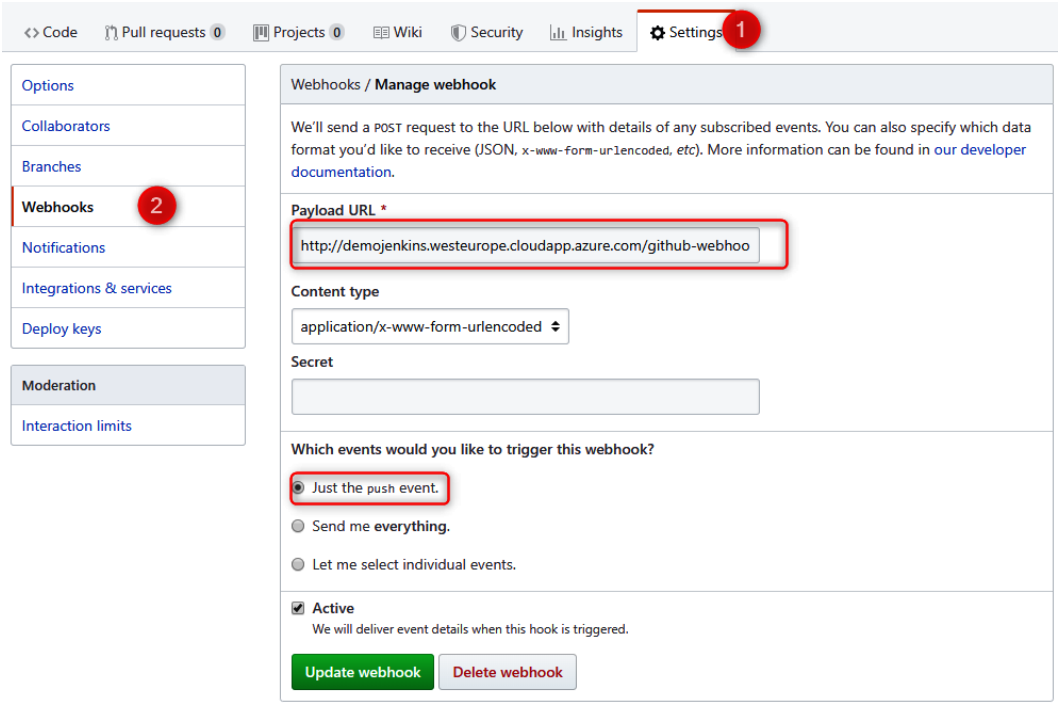

Figure 7.8 – GitHub webhook for Jenkins configuration

5. Finally, we can check on the GitHub interface that the webhook is well configured and that it communicates with Jenkins, as illustrated in the following screenshot:

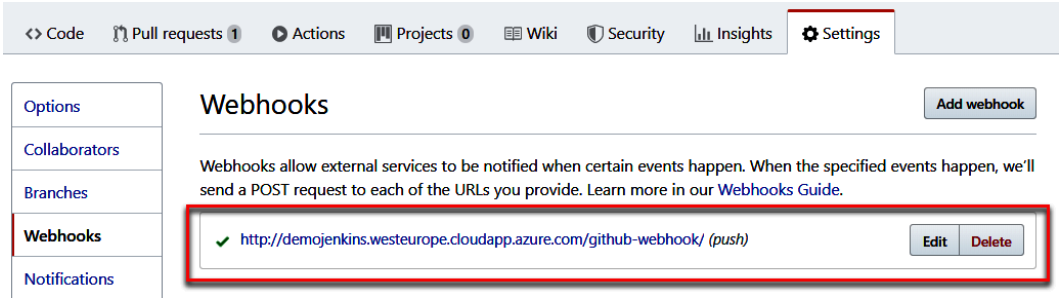

Figure 7.9 – GitHub Webhooks' validation

The configuration of GitHub is done. We will now proceed to create a new CI job in Jenkins.

Configuring a Jenkins CI job

To configure Jenkins, let's follow these steps:

1. First, we will create a new job by clicking on **New Item** or on the **create new jobs** link, as shown in the following screenshot:

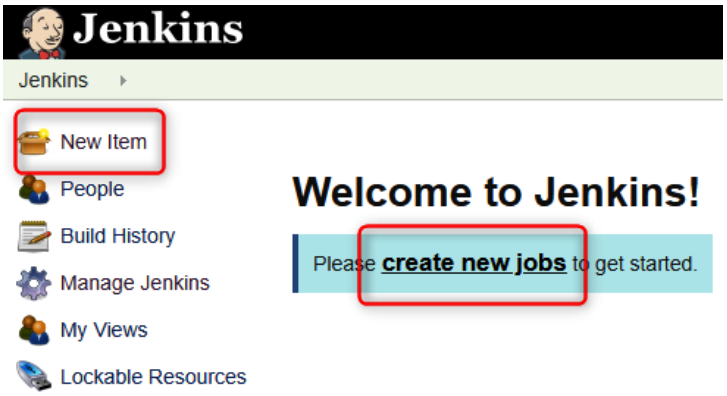

Figure 7.10 – Creating a new job in Jenkins

2. On the job configuration form, enter the name of the job—for example, `demoCI`—and choose the **Freestyle project** template, and then validate that by clicking on **OK**, as shown in the following screenshot:

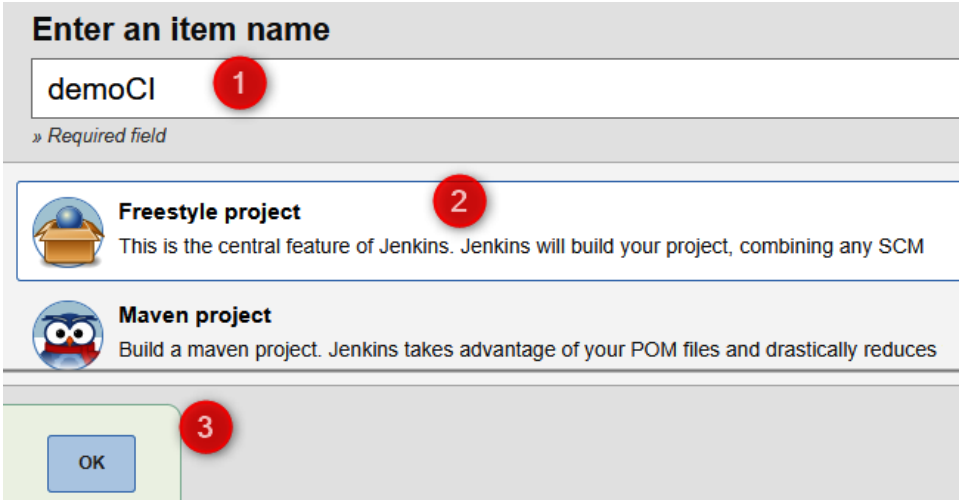

Figure 7.11 – Jenkins job name

3. Then, we configure the job with the following parameters:

   - In the **GitHub project** input, we enter the URL of the GitHub repository, as follows:

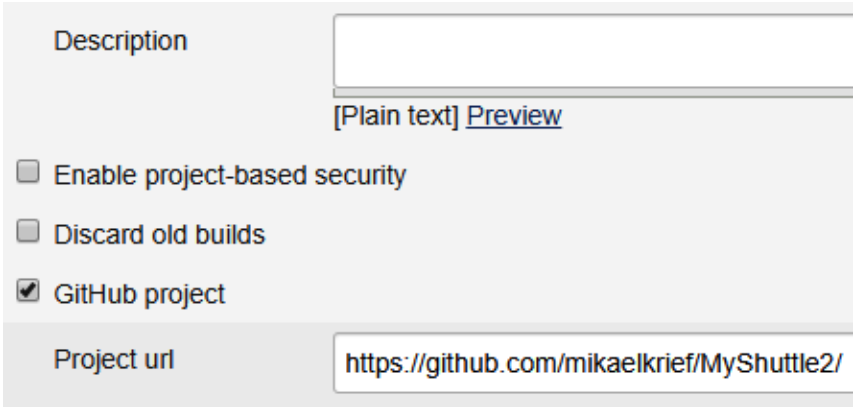

Figure 7.12 – Jenkins job: GitHub

   - In the **Source Code Management** section, enter the URL of the GitHub repository and the code branch, like this:

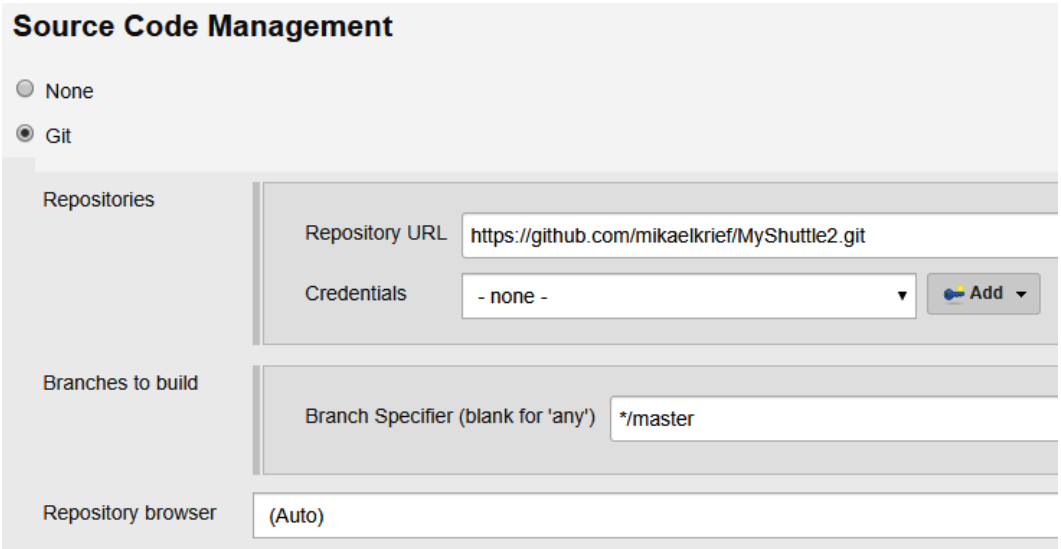

Figure 7.13 – Jenkins job: GitHub configuration

   - In the **Build Triggers** section, check the **GitHub hook trigger for GITScm polling** box, like this:

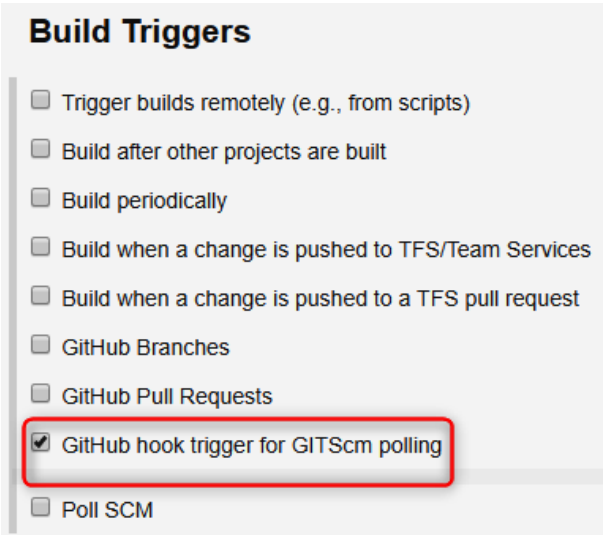

Figure 7.14 – Jenkins job: GitHub hook trigger

- In the **Build** section, in the **Add build step** drop-down list, we'll choose the **Execute shell** step for this lab. You can add as many actions as necessary to your CI (compilation, file copies, and tests). You can see some examples of possible actions in the following screenshot:

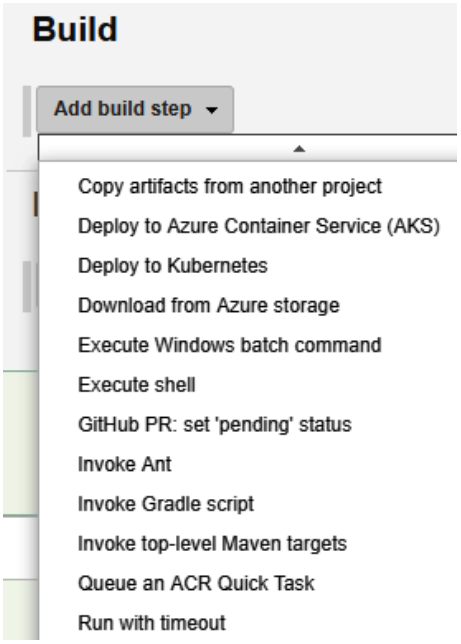

Figure 7.15 – Jenkins job: adding a build step

- Inside the textbox of the shell command, we enter the `printenv` command to be executed during the execution of the job pipeline, as illustrated in the following screenshot:

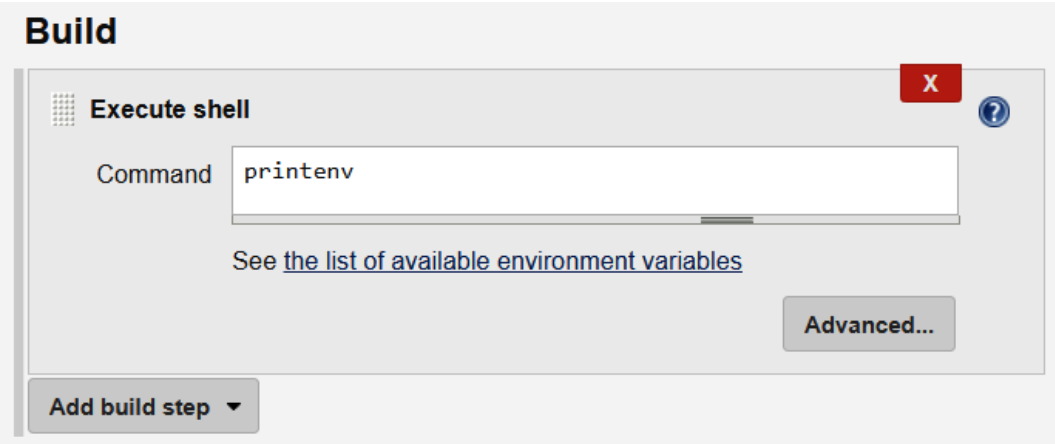

Figure 7.16 – Jenkins job command step sample

4. Then, we finish the configuration by clicking on **Apply** and then on the **Save** button.

Our Jenkins CI job has now been created and is configured to be triggered during a commit and to perform various actions.

We will now run it manually to test its proper functioning.

Executing a Jenkins job

To test job execution, we will perform these steps:

1. First, we will modify the code of our GitHub repository—for example, by modifying the `Readme.md` file.
2. Then, we commit to the master branch directly from the GitHub web interface.
3. What we see in Jenkins, right after making this commit, is that the `DemoCI` job is queued up and running.

The following screenshot shows the job execution queue:

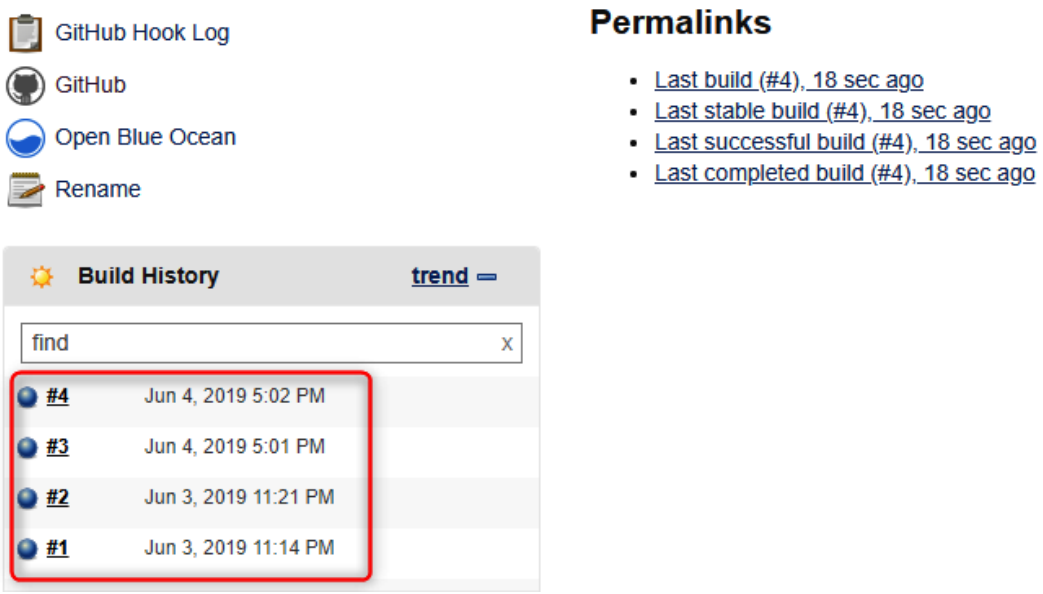

Figure 7.17 – Jenkins job run history

4. By clicking on the job, and then on the link of the **Console Output** menu, we can see the job execution logs, as shown in the following screenshot:

Figure 7.18 – Jenkins job console output

We have just created a CI job in Jenkins that runs during a commit of a Git repository (GitHub, in our example).

In this section, we have looked at the creation of a pipeline in Jenkins. Let's now look at how to create a CI/CD pipeline with another DevOps tool: Azure Pipelines.

In this section, we learned the basics of how to use YAML files that contain the definition of the pipeline and have many other very interesting features. To learn more, consult the documentation here: `https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops`.

Let's now look at creating a CI pipeline using GitLab CI.

Using GitLab CI

In the previous sections of this chapter, we learned how to create CI/CD pipelines with Jenkins and Azure Pipelines.

Now, let's look at a lab using another DevOps tool that is gaining popularity: **GitLab CI**.

GitLab CI is one of the services offered by GitLab (`https://about.gitlab.com/`), which, like Azure DevOps, is a cloud platform with the following attributes:

- A source code manager
- A CI/CD pipeline manager
- A board for project management

The other services it offers are listed here: `https://about.gitlab.com/features/`.

GitLab has a free price model with additional services that are subject to a charge; a price grid is available at `https://about.gitlab.com/pricing/`. The differences between Azure DevOps and GitLab are detailed in this link: `https://about.gitlab.com/devops-tools/azure-devops-vs-gitlab.html`.

In this lab, we'll find out about the following:

1. Authentication at GitLab
2. Creating a new project and versioning its code in GitLab
3. The creation and execution of a CI pipeline in GitLab CI

Authentication at GitLab

Creating a GitLab account is free and can be done either by creating a GitLab account or using external accounts, such as Google, GitHub, Twitter, or Bitbucket.

To create a GitLab account, we need to go to `https://gitlab.com/users/sign_in#register-pane` and choose the type of authentication.

The following screenshot shows the GitLab authentication form:

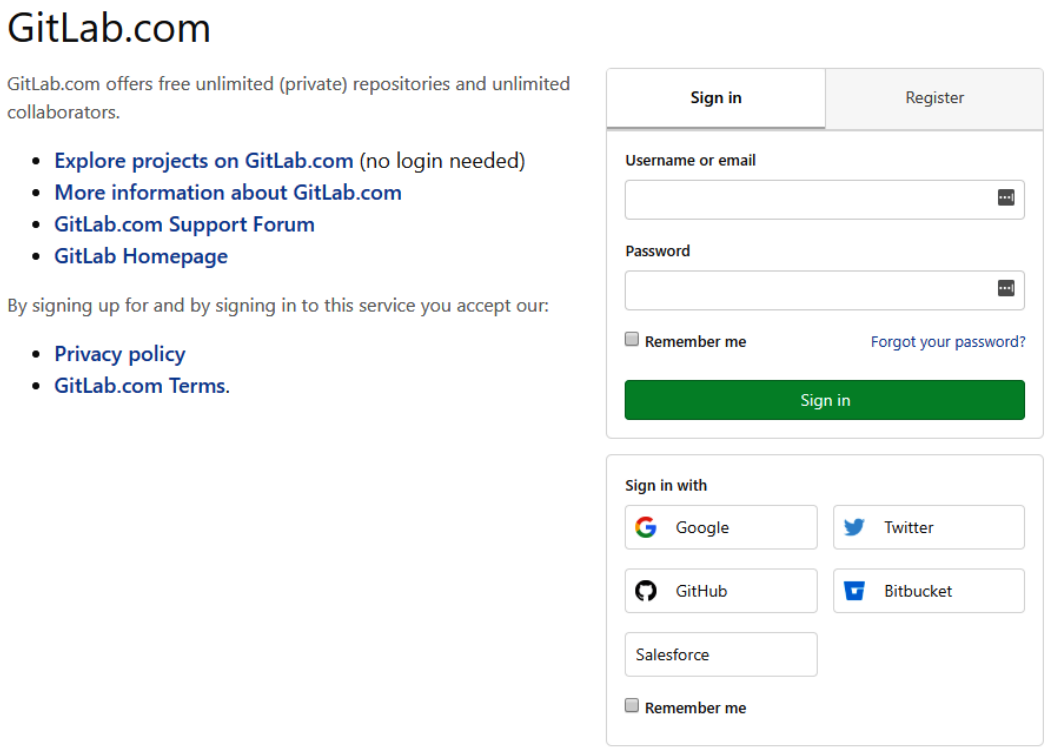

Figure 7.56 – GitLab registration

Once your account has been created and authenticated, you will be taken to the home page of your account, which offers all the functionalities shown in the following screenshot:

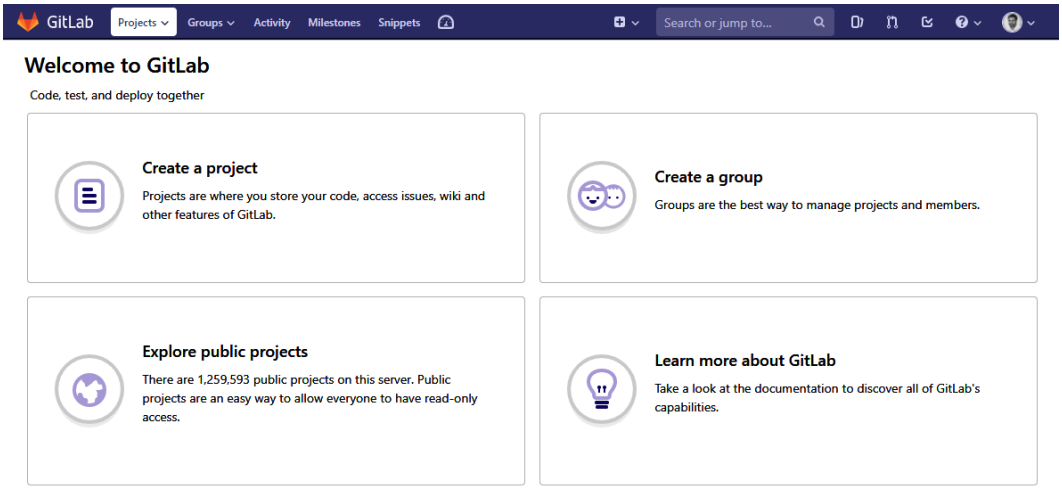

Figure 7.57 – GitLab home page

Now that we are done with authentication, let's go ahead and create a new project.

Creating a new project and managing your source code

To create a new project in GitLab, follow these steps:

1. Click on **Create a project** on the home page, as illustrated in the following screenshot:

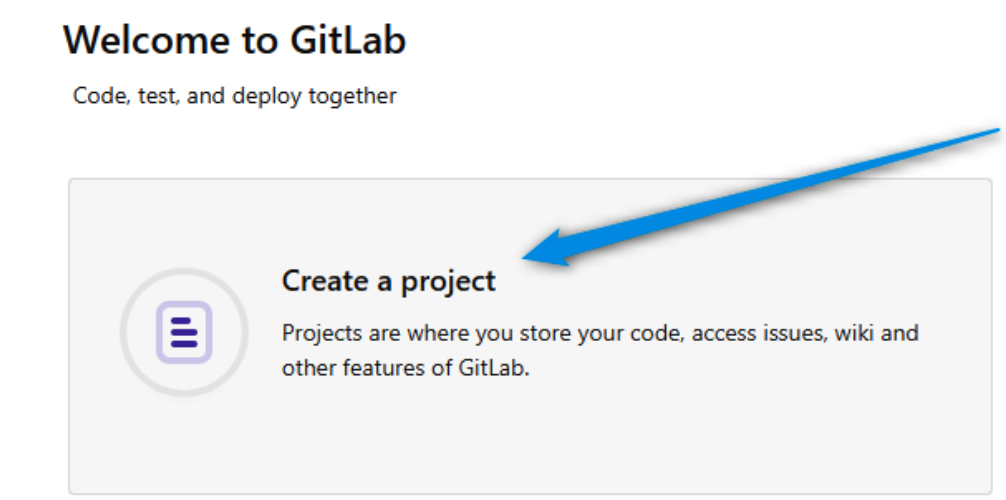

Figure 7.58 – GitLab new project

- Then, we can choose from a few options, as follows:
  - To create an empty project (without code), the form asks you to enter the project's name, as illustrated in the following screenshot:

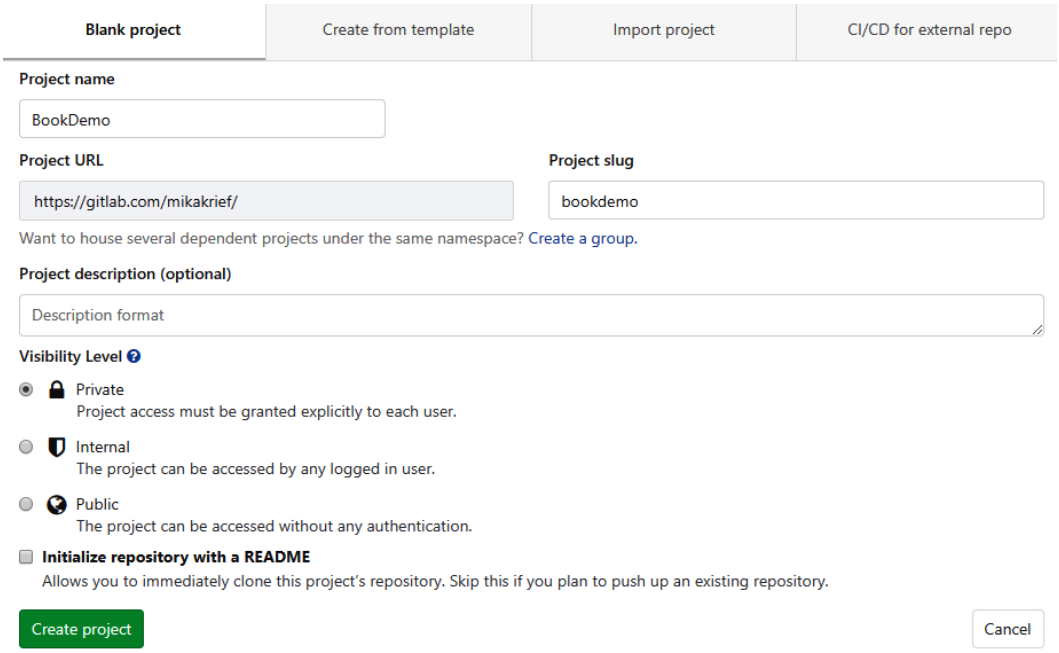

Figure 7.59 – GitLab project configuration

- To create a new project from a built-in template project, proceed as follows:

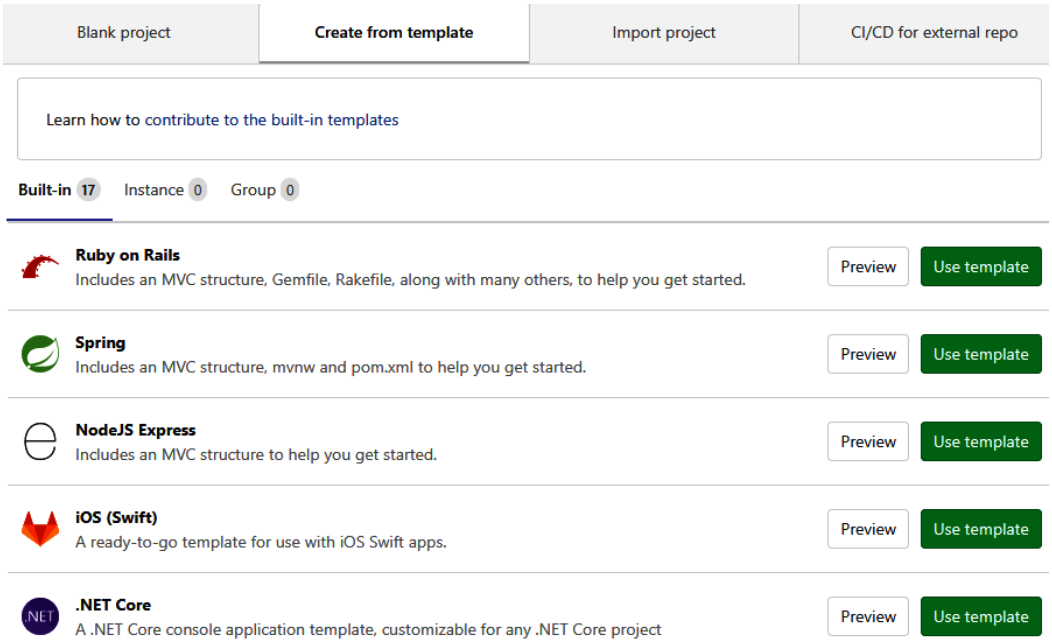

Figure 7.60 – GitLab project template

- To import code from an internal or external repository of another SVC platform, this is as shown in the following screenshot:

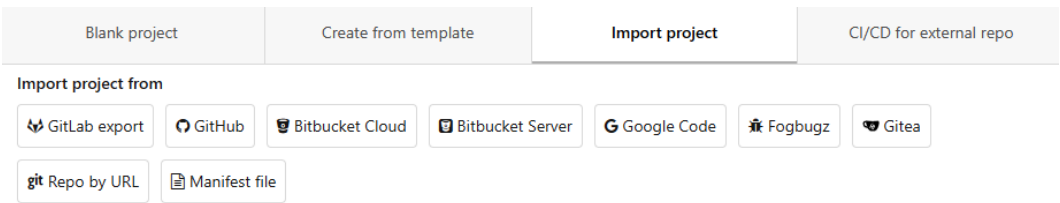

Figure 7.61 – GitLab: importing code

- The code to import is located in an external SVC repository, as shown in the following screenshot:

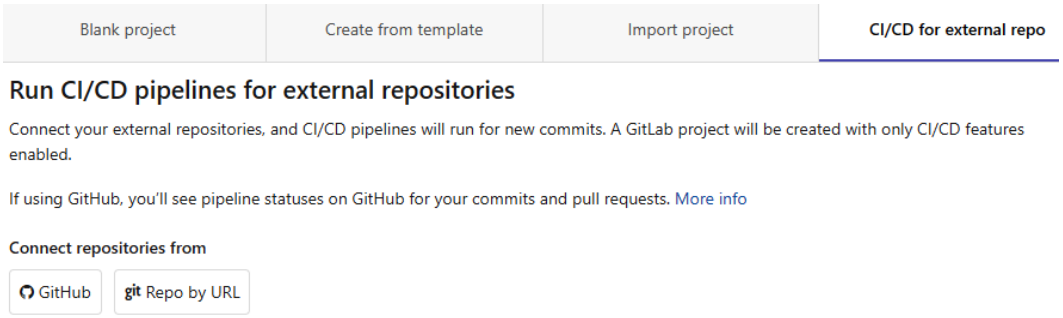

Figure 7.62 – GitLab: CI/CD for external repositories

In our case, for this lab, we will start with the first option, which is an empty project, and as we saw in the form, we choose a project name such as `BookDemo` and then validate it by clicking on the **Create a project** button.

2. Once the project is created, we'll have a page that indicates the different Git commands to execute to push its code.
3. To do this, on our local disk, we will create a new `gitlab-ci-demo.yml` file and then copy the content of our example, which can be found at `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP07`, in the `gitlab-ci-demo.yml` file.
4. Then, we will execute the following commands in a terminal to push the code into the repository, as seen in detail in *Chapter 6*, *Managing Your Source Code with Git*:

```
git init
git remote add origin <git repo Url>
git add .
git commit -m "Initial commit"
git push -u origin master
```

> **Note**
>
> During execution, an identification window will ask for your GitLab account **identifier** (**ID**) because it is a private project. Once logged in to your account on the GitLab web portal, your username will be available on your account page at `https://gitlab.com/profile/account`.

Once these commands have been executed, we'll obtain a remote GitLab repository with our lab code.

The following screenshot shows the remote GitLab repository:

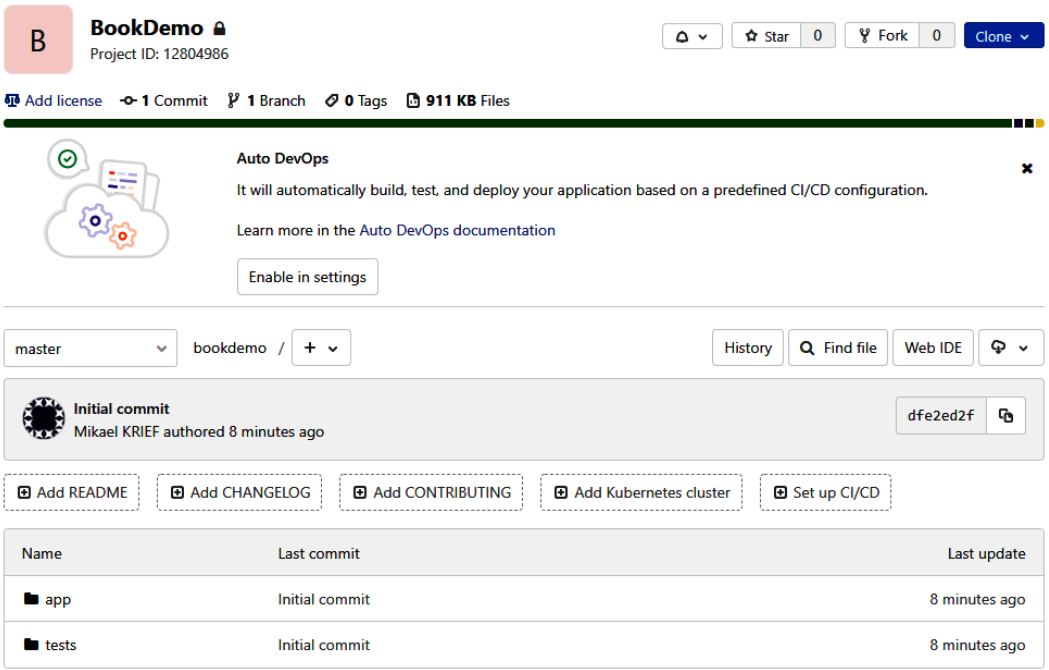

Figure 7.63 – GitLab repository

The code of our application has been deposited in GitLab, and we can now create our CI process with GitLab.

Creating a CI pipeline

In GitLab CI, the creation of a CI pipeline (and CD) is not done via a **graphical UI** (**GUI**), but with a YAML file at the root of the project.

This method, which consists of describing the process of a pipeline in a file that is located with the code, can be called **Pipeline as Code** (**PaC**), in the same way as **Infrastructure as Code** (**IaC**). We'll proceed as follows:

1. To create this pipeline, we will create, at the root of the application code, a `.gitlab-ci.yml` file with the following content:

```
image: microsoft/dotnet:latest
stages:
    - build
    - test
```

```
variables:
    BuildConfiguration: "Release"

build:
    stage: build
    script:
        - "cd app"
        - "dotnet restore"
        - "dotnet build --configuration
$BuildConfiguration"
test:
    stage: test
    script:
        - "cd tests"
        - "dotnet test --configuration
$BuildConfiguration"
```

> **Note**
>
> The code of this file is also available here: `https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/blob/main/CHAP07/gitlab-ci-demo.yml`.

We can see at the beginning of this code that we use a `microsoft/dotnet:latest` Docker image that will be mounted in a container and in which the actions of the pipeline will be executed.

Then, we define two stages, one for the build and one for the test execution, as well as a `BuildConfiguration` variable that will be used in the scripts.

Finally, we describe each of the stages of the scripts to be executed in their respective directories. These .NET core scripts are identical to the ones we saw in the *Using Azure Pipelines for CI/CD* section.

> **Note**
>
> Full documentation on the format and syntax of this `.gitlab-ci.yml` file is available here: `https://docs.gitlab.com/ee/ci/yaml/`.

2. Then, we will commit and push this file into the remote repository.
3. Just after pushing the code, we can see that the CI process has been triggered.

Our CI pipeline was, therefore, triggered when the code was pushed into the repository, so let's now look at how to see the details of its execution.

Accessing the CI pipeline execution details

To access the execution details of the executed CI pipeline, follow these steps:

1. In the GitLab CI menu, go to **CI / CD | Pipelines**, and you will see a list of pipeline executions, as shown in the following screenshot:

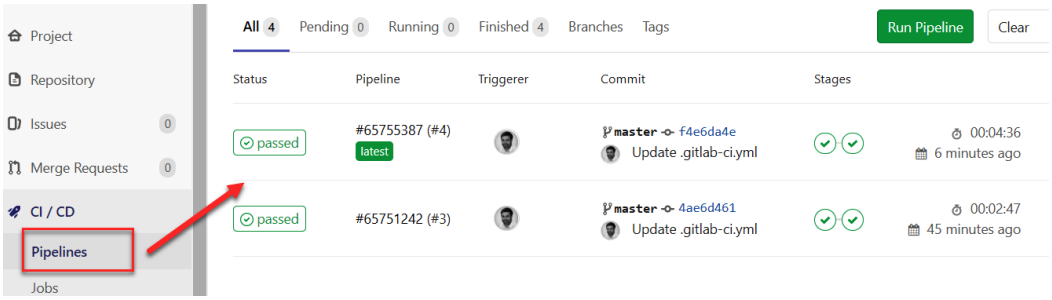

Figure 7.64 – GitLab pipelines

2. To display the details of the pipeline, we click on the desired pipeline execution, as illustrated in the following screenshot:

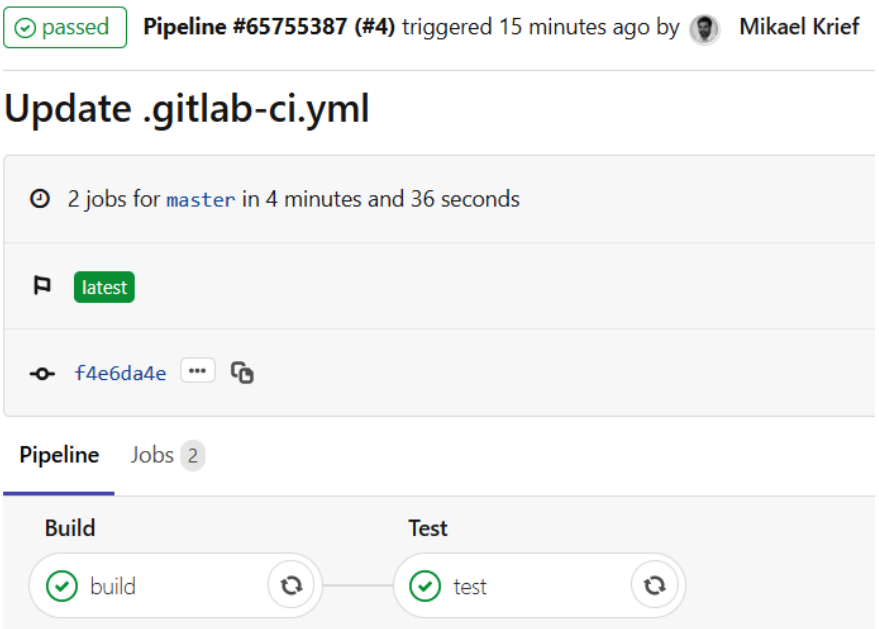

Figure 7.65 – GitLab pipeline execution

3. We can see the execution status, as well as the two stages that you defined in the pipeline YAML file. To view the details of the execution logs for a stage, we click on the stage, as shown in the following screenshot:

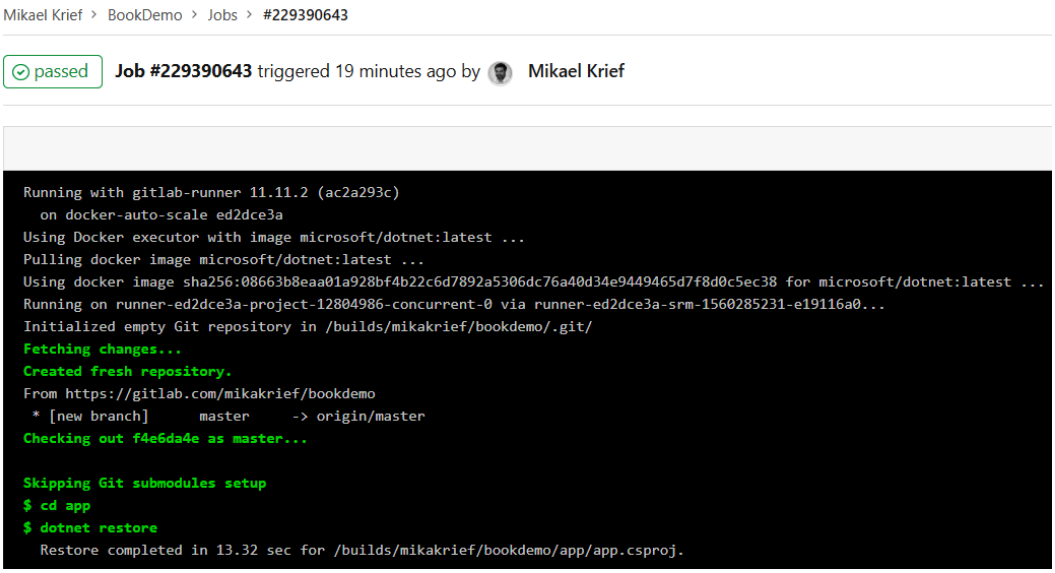

Figure 7.66 – GitLab execution log details

We can see the execution of the scripts written in the pipeline YAML file.

In this section, we have seen the implementation of a CI pipeline in GitLab CI with the initialization of a remote repository and the creation of a YAML file for configuring the pipeline as well as the execution of the pipeline.

Summary

In this chapter, we looked at one of the most important topics in DevOps: the CI/CD process. We started with a presentation of the principles of CI and CD. Then, we focused on package managers, looking at NuGet, npm, Nexus, and Azure Artifacts.

Finally, we saw how to implement and execute an E2E CI/CD pipeline using three different tools: Jenkins, Azure Pipelines, and GitLab CI. For each of them, we looked at the archiving of the application source code, along with the creation of a pipeline and its execution.

After reading this chapter, we should be able to create a pipeline for CI and CD with source code management as the source. In addition, we will be able to choose and use a package manager to centralize and distribute our packages.