Cloud-Native Continuous Integration and Delivery

Build, test, and deploy cloud-native applications in the cloud-native way

Onur Yilmaz

Packt
www.packt.com

CLOUD-NATIVE CONTINUOUS INTEGRATION AND DELIVERY

Author: Onur Yilmaz

Technical Reviewer: Hasan Turken

Managing Editor: Mahesh Dhyani

Acquisitions Editor: Aditya Date

Production Editor: Samita Warang

Editorial Board: David Barnes, Ewan Buckingham, Simon Cox, Manasa Kumar, Alex Mazonowicz, Douglas Paterson, Dominic Pereira, Shiny Poojary, Saman Siddiqui, Erol Staveley, Ankita Thakur, and Mohita Vyas

First Published: December 2018

Production Reference: 1201218

ISBN: 978-1-78980-565-9

Table of Contents

Preface

Cloud-NativeCI/CD Concepts

INTRODUCTION TO CLOUD-NATIVE CI/CD CONCEPTS

DEVOPS CULTURE

DEVOPS PRACTICES

DEVOPS TOOLCHAIN

CLOUD-NATIVE ARCHITECTURE

CLOUD-NATIVE APPLICATION CHARACTERISTICS

DEVOPS PATTERNS FOR CLOUD-NATIVE ARCHITECTURE

CHOOSING THE BEST CI/CD TOOLS

Cloud-Native Continuous Integration

UNIT TESTING

EXERCISE 3: PERFORMING UNIT TESTING FOR MICROSERVICES

SMOKE TESTING

EXERCISE 4: PERFORMING SMOKE TESTS FOR MICROSERVICES

INTEGRATION TESTING

EXERCISE 5: PERFORMING INTEGRATION TESTING FOR MICROSERVICES

BUILDING CLOUD-NATIVE APPLICATIONS

EXERCISE 6: CREATING MULTI-STAGE DOCKER BUILDS

CHECKLIST FOR CLOUD-NATIVE CI DESIGN

ACTIVITY 1: BUILDING A CI PIPELINE FOR CLOUD-

Cloud-Native Continuous Delivery and Deployment

Preface

About

This section briefly introduces the author, the coverage of this book, the technical skills you'll need to get started, and the hardware and software required to complete all of the included activities and exercises.

About the Book

When several developers work on the same code and do not merge their changes, the end result is a sure disaster. Cloud-native software development is a powerful tool to avoid this occurrence. However, cloud-native software development requires new ways of building and delivering applications. Specifically, operating in a continuous integration (CI) and continuous delivery (CD) environment is essential.

This book teaches you the skills you need to create a CI and CD environment for your applications, and deploy them using tools such as Kubernetes and Docker. By the end of this book, you'll be able to design professional and enterprise-ready CI/CD pipelines.

ABOUT THE AUTHOR

Onur Yilmaz is a software engineer at a multinational enterprise software company. He is a Certified Kubernetes Administrator (CKA) and works on Kubernetes and cloud management systems. He is a keen supporter of cutting-edge technologies including Docker, Kubernetes, and cloud-native applications.

OBJECTIVES

- Learn the basics of DevOps patterns for cloud-native architectures
- Learn the cloud-native way of designing CI/CD systems
- Create multi-stage builds and tests for Docker
- Apply the best practices for Docker container images
- Build and test applications on the cloud
- Learn how to continuously deliver to the Docker

registry

- Learn how to continuously deploy to Kubernetes
- Configure and deploy software to Kubernetes using Helm

AUDIENCE

This book is ideal for professionals who are interested in cloud-native software development. To benefit the most from this book, you should be familiar with developing, building, testing, integrating, and deploying containerized microservices on cloud systems. Basic proficiency in Git, Go, and Docker is required.

APPROACH

This book delivers its content through hands-on exercises. Throughout the book, you will learn about the required toolset by using on-premise, open source, and hosted cloud solutions. You'll find checklists, best practices, and critical points mentioned throughout the chapters, making things more interesting.

HARDWARE REQUIREMENTS

For an optimal student experience, we recommend the

following hardware configuration:

- Processor: Intel Core i5 or equivalent
- Memory: 4 GB of RAM or higher

SOFTWARE REQUIREMENTS

You'll also need the following software installed in advance:

- Sublime Text (latest version), Atom IDE (latest version), or another similar text editor application
- Docker
- Git

INSTALLATION AND SETUP

Before you start this book, we'll install Docker and Git, which are the tools used throughout this book. You will find the steps to install them here:

Installing Docker

Run the following commands on your system to install Docker.

```
curl -fsSL https://get.docker.com -o get-docker.sh

sh get-docker.sh
```

Installing Git

Please follow the steps for your operating system to install Git: https://docs.gitlab.com/ee/topics/git/how_to_install_git/.

INSTALLING THE CODE BUNDLE

Copy the code bundle for the class to the **`C:/Code`** folder.

CONVENTIONS

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create a **`build-push`** stage using the **`docker build`** and **`push`** commands using the **`$CI_COMMIT_SHA`** as commit-ID."

A block of code is set as follows:

```
FROM golang:1.11.2-alpine3.8 as builder

ADD . /go/src/gitlab.com/onuryilmaz/book-server-cd
```

```
WORKDIR /go/src/gitlab.com/onuryilmaz/book-server-cd/cmd

ARG VERSION

RUN go build -ldflags "-X main.version=$VERSION" -o book-server

FROM alpine:3.8 as production
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Open the GitLab interface and click the **CI/CD** tab and then click the **Run Pipeline** tab."

ADDITIONAL RESOURCES

The pipeline examples used in this book are hosted on GitLab at:

- https://gitlab.com/TrainingByPackt/blog-pipeline-example
- https://gitlab.com/TrainingByPackt/book-server
- https://gitlab.com/TrainingByPackt/book-server-cd

Additionally, the code bundle for this book is hosted on GitHub at https://github.com/TrainingByPackt/Cloud-Native-Continuous-Integration-and-Delivery.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Cloud-Native CI/CD Concepts

Learning Objectives

By the end of this chapter, you will be able to:

- Compare conventional approaches of software development with DevOps
- Describe the DevOps toolchain steps
- Identify the benefits of cloud-native architecture for software development
- Describe the DevOps patterns for a cloud-native environment
- Create a CI/CD pipeline in the cloud

This chapter introduces basic principles of DevOps and cloud-native approaches in addition to presenting the steps for creating a CI/CD pipeline in the cloud.

Introduction to Cloud-Native CI/CD Concepts

In the past few years, there have been several paradigm shifts in software development and operations. This has presented the industry with innovative methods for creating and installing applications. More importantly, two significant paradigm shifts have consolidated their capabilities for developing, installing, and managing scalable applications: DevOps and the cloud-native architecture. DevOps introduced a culture shift that increased focus on having smaller teams with agile development instead of large groups and long development cycles. Cloud-native microservices emerged with cloud-based horizontal scaling ability, thus providing service to millions of customers. With these two significant, powerful approaches combined, organizations now have the capability to create scalable, robust, and reliable applications with a high level of collaboration and information sharing among small teams. Before we begin expanding on DevOps and the cloud-native architecture, we will explore the limitations posed by conventional software development and how it compares with new approaches.

Conventional software development can be likened to manufacturing a passenger aircraft. The end product is enormous and requires considerable resources, such as a large infrastructure, capital, and personnel, to name a few. Requirement collections and planning are rigid and usually

take weeks to months to be finalized. Similar to this example, in the conventional development cycle, different parts of the software are always combined in the same product line that is specifically built for a monolithic bulky end product. Once developed, the product is finally delivered to customers. Note that there is very little flexibility presented to the customers in terms of how they wish to use the product's features or in terms of dynamically altering them as per market fluctuations.

Nowadays, software development can be likened to manufacturing drones. The goal is to have leaner end products that can be mass produced and distributed. Unlike bulky software, today, software is distributed as microservices and built with relatively smaller and sometimes even geographically distributed teams. Requirements collection and planning are more flexible, and it is possible to let customers decide how to use these services and configure them on the fly. In addition to maintenance, managing and updating the software-as-a-service is included in the life cycle. This revolution in software development is possible due to new cloud-based architectural approaches and DevOps related cultural changes in organizations.

It is challenging to develop and run the scalable cloud-native applications with tools and the mindset of conventional software development. Unlike large software packages that are delivered in disk drives and other storage devices, current

implementations are designed as microservices and packaged as containers that are available via cloud systems. These containers run in clusters that are managed by cloud providers such as **Amazon Web Services** (**AWS**) or **Google Cloud Platform** (**GCP**). Thus, organizations do not have to invest in costly servers or bother about having them run in-house. To develop, test, and run cloud-native applications, two essential DevOps practices must be implemented in a cloud-native manner: **continuous integration** (**CI**) and **continuous delivery**/**deployment** (**CD**).

Creating CI pipelines to test microservices and create containers is the first prerequisite of cloud-native application development. CD is based on delivering applications for customer usage and deploying them for production. With best practices, checklists, and real-life examples of cloud-native CI/CD pipelines, organizations can now bridge the gap between developers and customers efficiently and create reliable, robust, and scalable applications.

This chapter explores the impetus for the shift toward a DevOps culture and its impact on software development. Next, cloud-native architecture and essential characteristics of the applications of the future are listed. The chapter also explains how cloud-native architecture compliments DevOps and contributes toward successful organizations. DevOps practices for cloud-native architecture, namely CI and CD, are

described, and guidelines are provided for selecting the appropriate tools.

DevOps Culture

Software development organizations traditionally worked in a fast-paced environment without focusing on inter-team collaboration. Development teams attempted to produce software as soon as possible for deployment by the operations team. Without clear communication between development and operations, conflicts and product failures were inevitable. When organizations examined the problems in depth, they realized that development teams had almost no idea about the runtime environment. The operations team had practically no sound understanding of the requirements and features of the applications they were deploying. With enormous barriers between these teams, organizations created applications that did not simultaneously account for the runtime environment and software requirements. Consequently, neither development nor operations teams were held responsible for many problems and attempted to address several customer tickets, thus leading to the loss of many engineer hours and money.

DevOps—derived from Development Operations — culture came in to being to increase collaboration between development and operations. Organizations built DevOps

teams with engineers from development and operations backgrounds to eliminate the communication barrier between these groups. Besides, many practices and tools are implemented to increase automation and decrease the delivery times, and minimize the risks. Eventually, this culture shift in organizations fostered quality and reliability with reduced lead times. In these new teams, developers acknowledged operational knowledge such as cloud providers and customer environments.

Operations engineers also gained insight into the applications that they were deploying. Enhanced overall efficiency and advances in cloud-native architectures increased the adoption of DevOps culture in various level of organizations, from start-ups to enterprise companies.

Note

Although the term DevOps is used in various meanings, job postings, and company culture manifestos, there is not still one accepted academic or practical definition. The DevOps term was coined by Patrick Debois in 2009 and first used in the DevOpsDays Conference that started in Belgium.

To summarize, we first described the issues encountered in the conventional method of software development. We discussed how DevOps increases collaboration and mitigates problems that are encountered in conventional approaches for software development. In the next section, we will discuss the

best practices for implementing DevOps.

DEVOPS PRACTICES

Organizations adopt unique methods to implement DevOps. Thus, there are no specific standards in terms of implementation practices. In other words, it is difficult to find a single approach with regard to implementing a DevOps culture shift when considering unique product requirements and organizational structures. However, there are certain core best practices that have been implemented in the industry by successful companies. The following ideas cover the core the DevOps philosophy:

- **Continuous Integration** (**CI**): Continuous integration focuses on integrating changes from different sources as soon as possible. Integration covers building, reviewing code quality, and testing. The main idea of CI is finding bugs and conflicts as quickly as possible and solving them early in the software life cycle.
- **Continuous Delivery** (**CD**): Continuous delivery focuses on delivering and packaging the software under test as soon as possible. Similar to CI, CD aims to create production-ready packages and deploy them to the production environment. With this idea, all changes will be in the service of customers, and developers will be able to see their recent commits live.

- **Monitoring and logging**: Monitoring metrics and collecting application logs is critical for investigating the causes of problems. Creating notifications over parameters and active control of systems help to create a reliable environment for end users. One of the most crucial points is that monitoring could create a proactive path for finding problems rather than waiting for customers to encounter issues and raise tickets.
- **Communication and collaboration**: The communication and collaboration of different stakeholders is crucial to success in DevOps. All tools, procedures, and organizational changes should be implemented to increase communication and cooperation. Knowledge and information sharing with open communication channels between teams enables transparency and leads to successful organizations.

Until now, we have discussed the best practices to implement the DevOps approach. In the next section, we will describe how the DevOps tools chain in conjunction with the aforementioned best practices lead to the creation of a value chain.

DEVOPS TOOLCHAIN

The DevOps toolchain enlists practices that connect development and operations teams with the aim of creating a

value chain. The main stages of the chain, along with their interconnectivity, are presented as follows:

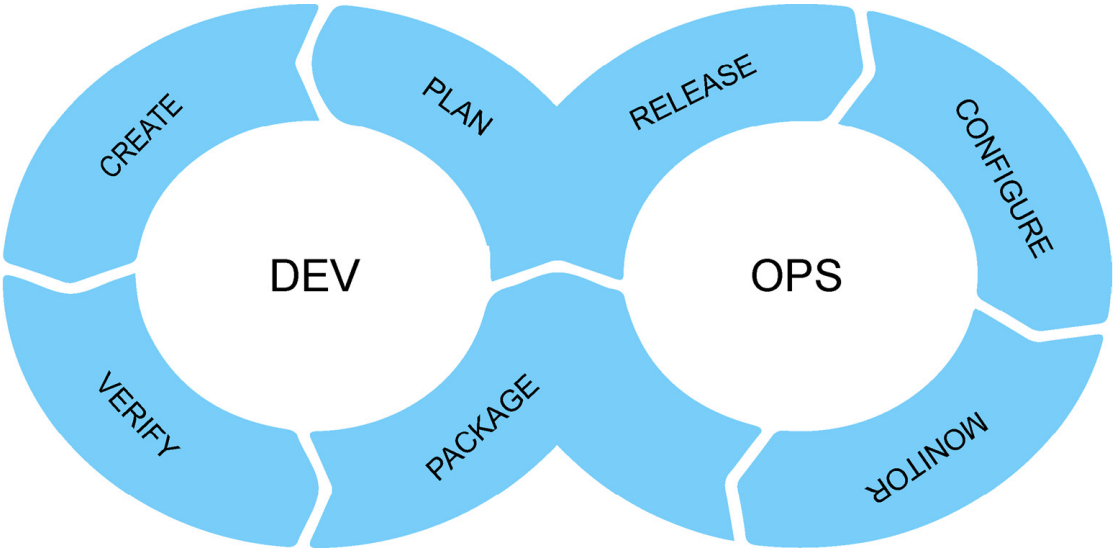

Figure 1.1: The DevOps toolchain

The inputs and outputs of each stage are presented in the following flow chart:

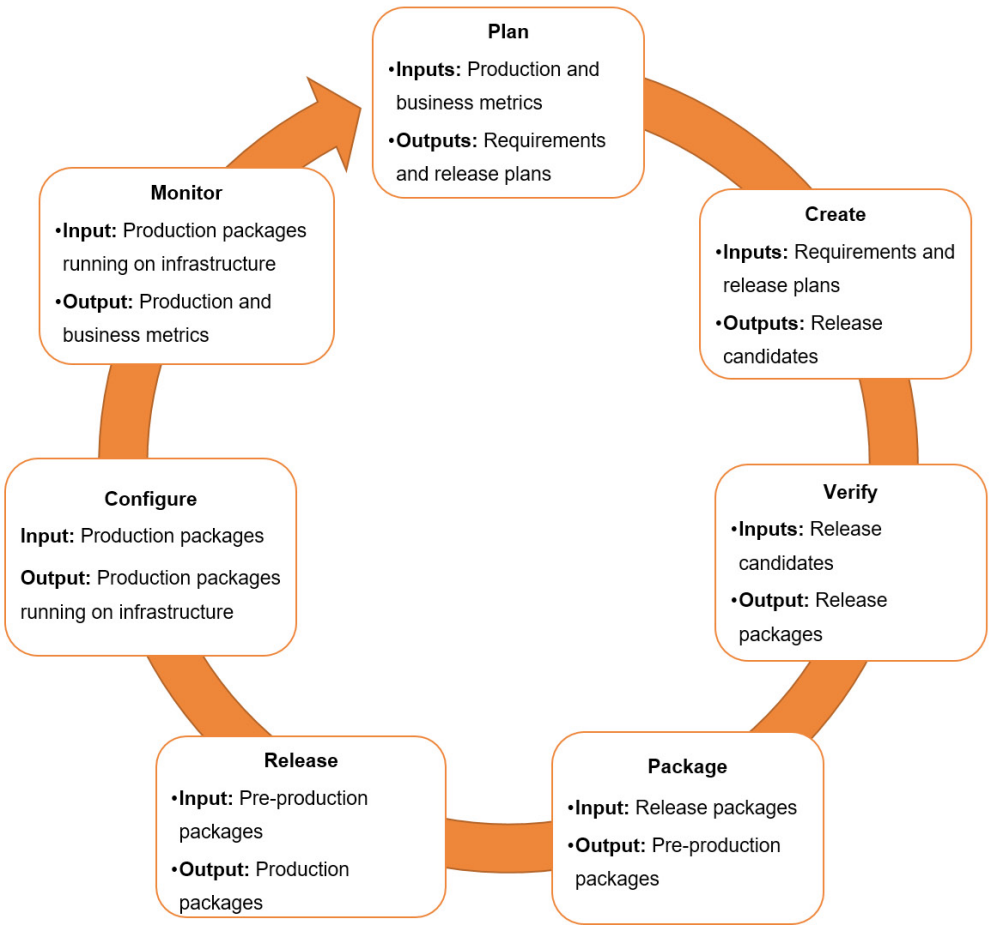

Figure 1.2: Detailed steps of the DevOps toolchain

When a new project is on the table, the chain originates from planning and then progresses to creating the software. The next steps are verification and packaging. Completion of packaging marks the end of the development phase. Thereafter, operations begin from release, followed by configuration. Any feedback and insights obtained at the end of monitoring feed in to the development phase again, thereby completing the cycle. It is important not to skip any part of the toolchain and create an environment where processes feed each other with complete data. For instance, if monitoring fails to provide accurate information about the production environment, development may not have any idea about the outages in production. The development team will be under the false impression that their application is running and scaling with customer demand. However, if the monitoring feeds planning with accurate information, development teams could plan and fix their problems in scaling. As DevOps tries to remove the barriers between development and operations, meticulous execution of each stage in the DevOps tool chain is crucial. The most natural and expected benefits of DevOps can be summarized as follows:

- **Speed**: The DevOps model and its continuous delivery principles decrease the time to deliver new features to the market.
- **Reliability**: Continuous integration and testing

throughout the product's life cycle helps to increase reliability of products. With metrics collected by monitoring systems, applications evolve to be more stable and reliable.

- **Scalability**: Not only software but also infrastructure is managed as code in the DevOps environment. This approach makes it easier to manage, deploy, and scale with customer demand.

DevOps culture, with its best practices and toolchains, provides many benefits to organizations. However, before implementation, understanding the current company's culture and creating a feasible action plan for introducing DevOps is crucial. In the following sections, how DevOps practices are implemented for applications and introduction to cloud-native architecture is explained in more detail.

Cloud-Native Architecture

Cloud-native application development focuses on building and running applications that utilize the advantages of cloud services. This focus does not mandate any specific cloud provider to deploy the applications; however, it concentrates on the approach of development and deployment. It consists of creating agile, reusable, and scalable components, and deploying them as resilient services in dynamically orchestrated cloud environments. Cloud-native services are

essential since they serve millions of customers daily and empower social networks such as Facebook, Twitter, online retailers such as Amazon, real-time car-sharing applications such as Uber, and many more.

Note

For more comprehensive information on cloud-native technologies, please refer to the following link: https://github.com/cncf/toc/blob/master/DEFINITION.md.

Knowing the differences between conventional and cloud-native approaches is essential. Specifically, conventional and cloud-native application development can be distinguished through the following four views:

- **Focus**: Conventionally, applications are designed for long lifespans that are included with years of maintenance agreements. However, cloud-native applications focus on how quickly applications can be market-ready with flexible subscription agreements.
- **Team**: Conventional software teams work independently of each other and focus on their specified areas, such as development, operations, security, and quality. In contrast, cloud-native applications are collaboratively developed and maintained by DevOps teams that comprise members focusing on different areas.
- **Architecture**: Monolithic applications and the firmly

coupled dependencies between them are the mainstream architectural approaches of conventional software. An example of monolithic design could be an e-commerce website where four different software components, namely the user **`frontend`**, **`checkout`**, **`buy`**, and **`user promotions`** are packaged as a single Java package and deployed to production. Each of these components may have several different functions aimed at addressing certain objectives of the website. All components call each other via function calls, and thus they are strictly dependent on each other. For instance, while creating an invoice, the **`buy`** package will directly call, for example, the **`CreateInvoice`** function of the **`checkout`** package. On the contrary, cloud-native applications are loosely coupled services communicating over defined APIs. When the same e-commerce website is designed with loosely coupled components, all of the components can call each other over API calls. With this approach, the **`buy`** package will create a **`POST`** request to a REST API endpoint, for example, **`/v1/invoice`**, to create the invoice.

- **Infrastructure**: Conventional applications are installed on and deployed through large servers that have been configured according to the end user environment. On the contrary, cloud-native applications run as containers and are ready to run, irrespective of vendor-specific requirements. Besides, capacity planning is for peak demand in traditional software systems; however, cloud-

native applications are run on a scalable, on-demand infrastructure.

In addition to the comparison with conventional software development, there are more characteristics of the cloud-native architecture. In the following section, all key cloud-native architecture characteristics are explained in depth. Note that most features have emerged with cloud-native applications and have changed the method of software development.

CLOUD-NATIVE APPLICATION CHARACTERISTICS

Characteristics of cloud-native applications can be grouped into three categories: **design**, **development**, and **operations**. These groups also indicate how cloud-native architectures are implemented throughout the life cycle of software development. First, design characteristics focus on how cloud-native applications are structured and planned. Then, development characteristics focus on the essential points for creating cloud-native applications. Finally, operations concentrate on the installation, runtime, and infrastructure features of cloud-native architecture. In this section, we will discuss all three characteristics in detail.

Design: Design is categorized into microservices and API communication:

- **Microservices**: Applications are designed as loosely coupled services that exist independent independently of each other. These microservices focus on a small subset of functionalities and discover other services during runtime. For instance, frontend services and backend services run independently, and the frontend finds the IP address of the backend from service discovery to send queries. Each service focuses only on its functionalities and does not directly depend on another service.
- **API Communication**: Services are designed to use lightweight API protocols to communicate with each other. APIs are versioned, and services interact with each other without any inconsistency. For instance, the frontend service reaches the backend via a REST API, and all API endpoints are versioned. For example, consider a versioned endpoint API: `/v1/orders`. When the backend is updated and changes its REST API, the endpoints will start with `v2` instead of `v1`. It ensures that the frontend still works with `v1` endpoints until it gets updated to work with `v2` endpoints without any inconsistency.

Development: Development is categorized into most suitable programming language and light weight containers:

- **Most Suitable Programming Language**: Cloud-native applications are developed using a programming

language and framework that is most suited for its functionality. It is aimed to have various programming languages working together while exploiting their best features. For instance, REST APIs could be developed in **Go** for concurrency, and the streaming service could be implemented in **Node.js** using **WebSockets**. Frontend services are not required to know the implementation details, as long as the programming languages implement the expected APIs.

- **Lightweight containers**: Cloud-native applications are packaged and delivered as containers. Each container consists of minimum requirements, such as operating system and dependency libraries of the service, in order to be as lightweight as possible. Containers enable scalability and encapsulate microservices for efficient management. For instance, the frontend and its **JavaScript** libraries create a **Docker** container, whereas the backend and its database connectors create another Docker container. Each service is packaged, self-sufficient, and can be scaled easily without knowing the internals of services.

Operations: Operations in categorized into isolation, elastic cloud infrastructure, and automation:

- **Isolation**: Cloud-native applications are isolated from their runtime and operating system dependencies, as well as those of other applications. This feature enables service

portability without making any further modifications. For instance, the same container image of the frontend service could simultaneously run on the laptop of the developer for testing new features and on **AWS** servers to serve millions of customers.

- **Elastic Cloud Infrastructure**: Cloud-native applications should run on a flexible infrastructure that could expand with usage. These flexible infrastructures are public or on-premise cloud systems that are shared by multiple services, users, and even companies, to achieve cost efficiency.
- **Automation**: Cloud-native applications and their life cycles should be automated as much as possible. Every step of development and deployment, such as integration, testing, provisioning of infrastructure, monitoring for capability, log collection, and auto-scaling and alerting, needs automation. Automation is crucial for reliable, scalable, and resilient cloud-native applications. Without automation, there are numerous manual steps to provision the infrastructure, configure the applications, run them, and check for their statuses. All of these manual steps are prone to human error, and it is unlikely to create reliable and robust systems that are scalable.

In this section, we saw the basic characteristics of cloud-native applications. In the next section, we will see how cloud-native architectures and the DevOps culture complement each

other to yield successful application development and deployment.

DevOps Patterns for Cloud-Native Architecture

Both DevOps culture and the cloud-native architecture complement each other in their contribution to bring about a change in software development and in making companies successful. While DevOps focuses on collaboration and the fast delivery of applications, the cloud offers scalability and automation tools to deliver applications to customers. In this section, we will discuss how the DevOps culture and cloud-native architecture work in sync and can be practically implemented.

DevOps attempts to eliminate time and resource wastage in software development by increasing automation and collaboration. Cloud-native application development focuses on building and running applications that utilize the advantages of cloud services. When they melt in the same pot, the cloud-native architecture and cloud-cloud computing enable and adopt DevOps culture in two main directions:

- **Platform**: Cloud-computing provides all platform requirements for DevOps processes such as testing, development, and production. This enables organizations

to smoothly run every step of the DevOps toolchain on cloud platforms. For instance, verification, release, and production stages of the DevOps toolchain can be easily placed on separate **Kubernetes** namespaces in **GCP** with complete isolation.

- **Tooling**: The DevOps culture focuses on automation, and it needs reliable and scalable tooling for continuous integration, deployment, and delivery. To exploit the scalability and reliability that's provided by cloud platforms, these tools inevitably have to be cloud-native. For instance, **AWS Code build**, which is a continuous code build tool by **AWS**, is widely used for a reliable, managed, and secure method of testing and integrating applications.

Not only solo DevOps or cloud-native applications, but also their combination has changed software development. In the following table, fundamental changes are briefly summarized:

	Pre-DevOps	DevOps
Product	Software is packaged and delivered.	Rather than individual software packages, managed services are run.
Development	Development ceases when features are completed.	Devolopment is an ongoing process as long as sevices are running.
Teams	Each team focuses on their own domain.	All teams collaborate.
Monitoring	Operation team tracks the metric production.	Operation teams provide tools for tracking.
Deployment	Each customer own their individual envioronment.	Customer share the same infrastructure via virtual isolation.

Figure 1.3: Differences between Pre-DevOps and DevOps approaches

Today, and presumably in the future, organizations would not only deliver software but also provide services. Similar to the **Software-as-a-Service** (**SaaS**) products of today, the approach will be more mainstream, and microservices-as-a-service products will spread throughout the market. To deliver and manage cloud-native services of the future, you will need to implement DevOps practices. The most critical cloud-native DevOps practices are continuous integration and continuous delivery/deployment. Each of these will be briefly discussed as follows:

- **Continuous Integration** (**CI**): This practice concentrates on integrating the code several times a day, or more commonly, with every commit. With every commit, a hierarchy of tests are run, starting from unit tests to integration tests. Also, software executables are built to check inconsistencies between libraries or dependencies. With validation provided by the test and build results, success or failures indicate whether the codebase works. Tests and builds of the CI could run on on-premise systems or cloud providers. CI systems are expected to be always up and running and be on the lookout for the future commits from developers.
- **Continuous Delivery** (**CD**): CD is an extension of CI to automatically deliver or deploy packages that have been validated by CI. CD focuses on creating and publishing

software packages and Docker containers in the cloud-native world automatically with every commit. CD focuses on updating applications on the customer side or public clouds with the latest packages automatically. In this book, both continuous delivery and deployment topics are covered and abbreviated as CD.

In the next section, we will explore the essential characteristics of CI/CD tools in order to help you choose the right subset for your organization.

Choosing the best CI/CD tools

The DevOps culture and the practices of CI/CD require modern tools for building, testing, packaging, deploying, and monitoring. There are many open source, licensed, and vendor-specific tools on the market with different prominent features. In this section, we will first categorize CI/CD tools and then present a guideline for choosing the appropriate ones.

DevOps tools can be categorized as follows, starting from source code to the application running in a production environment:

- Version Control Systems: GitHub, GitLab, and Bitbucket
- Continuous Integration: Jenkins, Travis CI, and

Concourse

- Test Frameworks: Selenium, JUnit, and pytest
- Artifact Management: Maven, Docker, and npm
- Continuous Delivery/Deployment: AWS Code pipeline, Codefresh, and Wercker
- Infrastructure Provisioning: AWS, GCP, and Microsoft Azure
- Release Management: Octopus Delivery, Spinnaker, and Helm
- Log Aggregation: Splunk, ELK stack, and Loggly
- Metric Collection: Heapster, Prometheus, and InfluxData
- Team Communication: Slack, Stride, and Microsoft Teams

On the market, there are plenty of tools with robust features that will make the tool qualified for more than one of the preceding categories. To select an appropriate tool, considering the pros and cons of each is difficult, owing to the uniqueness of organizations and software requirements. Therefore, the following guidelines could help to evaluate the core features of the tools within a continuously evolving industry:

Note

No Silver Bullet—Essence and Accident in Software Engineering was written by Turing Award winner Fred Brooks in 1986. In the paper, it is argued that "There is no single development, in either technology or management technique, which by itself promises even one order of magnitude (tenfold) improvement within a decade in productivity, in reliability, in simplicity." This idea is still valid for most software development fields due to complexity.

Enhanced collaboration: To have a successful DevOps culture in place, all of the tools in the DevOps chain should focus on increasing collaboration. Although there are specific tools, such as **Slack**, that have cooperation as the main focus, it is crucial to select tools that improve collaboration for every step in software development and delivery. For instance, if you need a source code versioning system, the most basic approach is to set up a bare `git` server with a single line of code: `sudo apt-get install git-core`.

With that set up, all of the components will be required to use `git` command-line tools to interact with the `git` server. Also, the team will carry the discussions and code reviews to other platforms, such as emails. There are tools such as **GitHub**, **GitLab**, or **Bitbucket** that integrate code reviews, pull requests, and discussion capabilities. Everyone on the team can quickly check the latest pull requests, code reviews, and issues. This eventually increases collaboration. Usage

experience differences can be checked out in the following screenshots, between the bare `git` server with a command-line interface and **GitLab** with the merge request screen. The crucial point is that both `git` server setup and **GitLab** solve the source code versioning problem, but they differ on the level of collaboration increase.

One of the key points while evaluating tools is taking the collaboration capabilities of the tools into consideration even if all the tools provide the same main functionality. The following screen shot shows how **GitLab** provides a better collaborative experience in comparison to a bare `git` server:

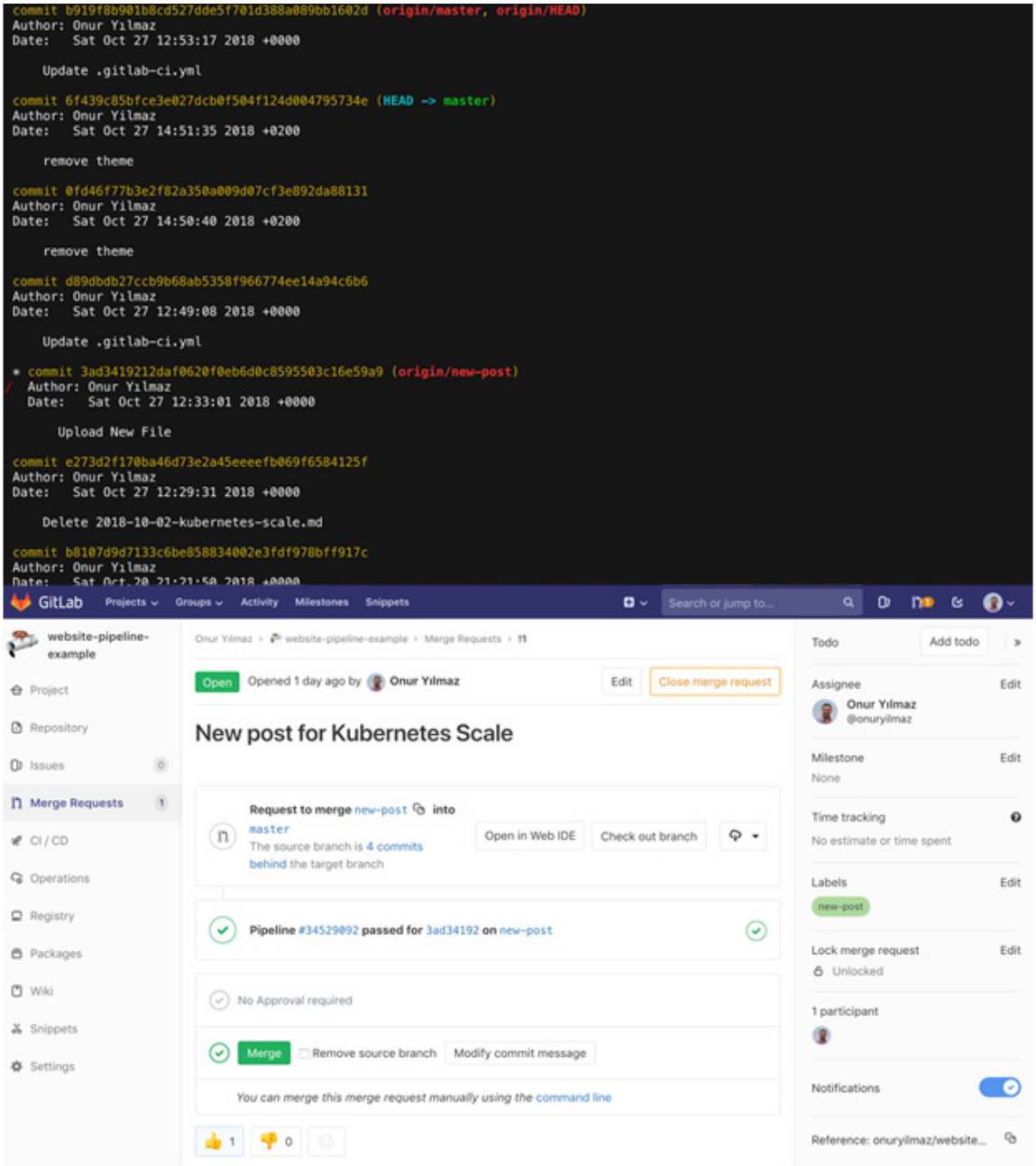

Figure 1.4: CLI for git server versus GitLab merge request

API integration: The DevOps toolchain and its operations need a high level of automation and configuration. It is unlikely that you should need to hire people to configure infrastructure for every run of the integration test. Instead, all stages of DevOps, from source code to production, are required to expose their APIs. It enables applications to communicate with each other, sending build results, configuration, and artifacts. Rich API functionality enables

new tools to be plugged in to the toolchain and work without high customization. Therefore, exposing the APIs is not only crucial for the first setup of the DevOps toolchain, but is also crucial when new tools replace old ones. Accordingly, the second of the key points while evaluating tools is API exposure and the composition of APIs to create a value chain.

Learning curve: Cloud-native DevOps tools try to follow up the latest cloud industry standards and vendor-specific requirements. Besides, most tools focus on user experience and are open to extensions with custom plugins at the same time. These different features could make DevOps tools complicated for beginners. Although there might be experienced users in the team, it is vital to select tools that entail exponential learning curves, starting from zero experience. In other words, tools should allow users to gain competency in a short amount of time, as illustrated in the following diagram. There are three key points to check for a tool that is suitable for everyone in the team:

Documentation: Official documentation, example applications, and references are essential to learn and gain competency.

Community and Support: Online and offline communities and support are critical for solving problems, considering the broad scope of DevOps tools and cloud integrations.

Multiple Access: Selection of the appropriate tools and,

having different methods of access, such as API, web interface, and CLI, is essential. This enables beginners to discover tools using the web interface, while experienced users can automate extensively and configure using the API and command line:

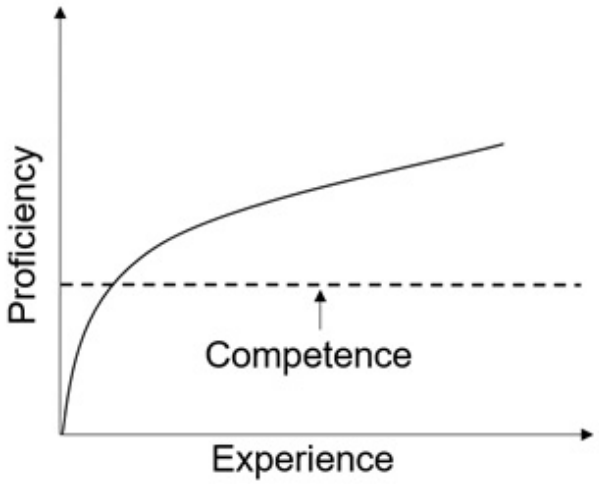

Figure 1.5: Exponential learning curve to a competence limit

DevOps practices for the cloud-native world can be viewed as a voyage in which both naive and experienced sailors are in the same boat. All parties need to get some competency, and selected tools should allow for this with their learning curves. Thus, this is the third crucial point while choosing a tool for an organization.

Three main points of enhanced collaboration, API exposure, and learning curves are the must-have features of DevOps tools and also important measures for comparing tools over each other. In the next section, the most popular cloud-native DevOps tools are evaluated using the guideline points mentioned.

EXERCISE 1: BUILDING, DEPLOYING, AND UPDATING YOUR BLOG IN THE CLOUD

Blog websites are popular in areas such as company websites, technology news, or sharing personal journeys. In this exercise, we aim to create a blog where the contents are written and kept in the source code repository, and the site is generated and automatically published by CI/CD pipelines in the cloud.

An up and running live blog website with its contents should appear as follows:

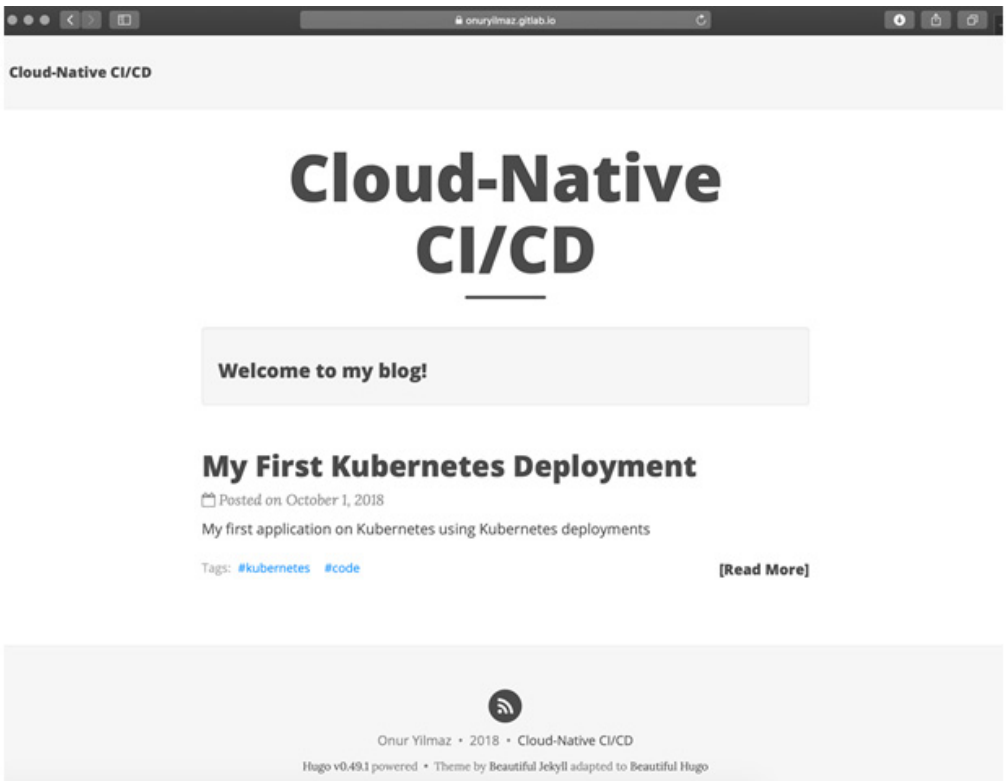

Figure 1.6: A sample blog with a single post

When a new blog post is added to the source code repository, the site should automatically be updated with the new material:

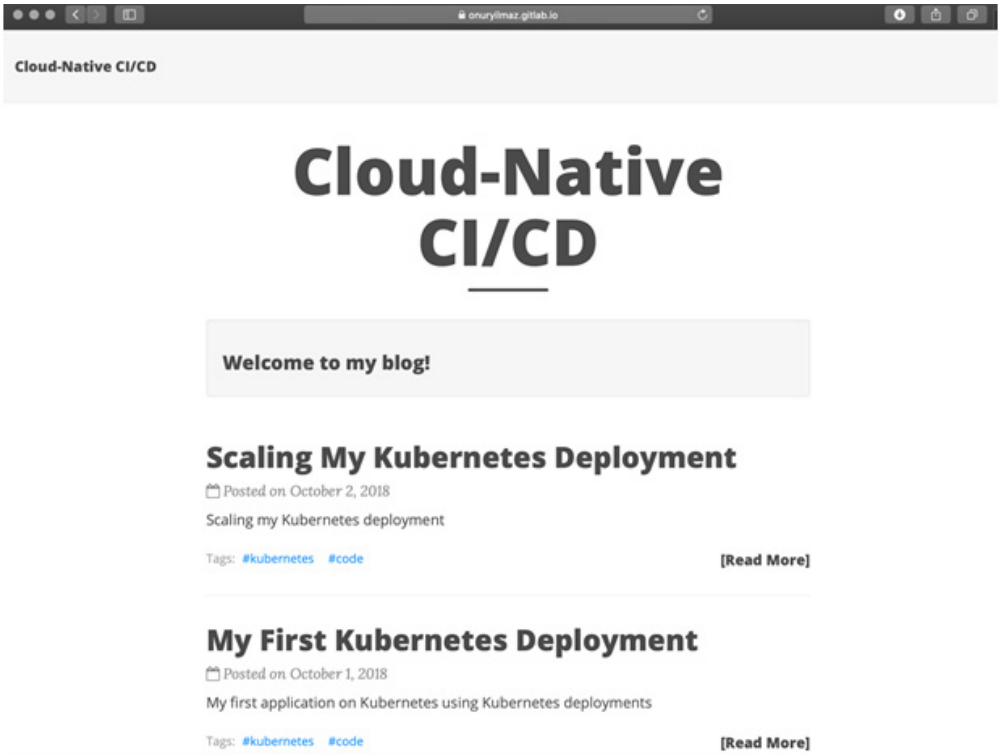

Figure 1.7: An updated blog with a new post added

Before we begin this exercise, ensure that the following prerequisites are satisfied:

- The blog and CI/CD pipelines are running on managed cloud services. In this book, we are using **GitLab**. Thus, you need to ensure that you have a **GitLab** account to which the projects provided in the book can be forked. You can sign up for an account at https://gitlab.com/.
- Website content, namely the HTML files, are generated by **Hugo** (https://gohugo.io/), which is a framework for building websites. Styling of the blog is handled with **beautifulhugo** (https://github.com/halogenica/beautifulhugo), which is a Hugo template repository.
- The website's pipeline example repository

(https://gitlab.com/TrainingByPackt/blog-pipeline-example) should be forked to your individual accounts.

Note

The code files for this exercise can be found here: https://bit.ly/2PBKisL

To successfully complete this exercise, we need to ensure that the following steps are executed:

1. Fork the project on **GitLab** to your namespace by clicking the forking icon, as shown in the following screenshot:

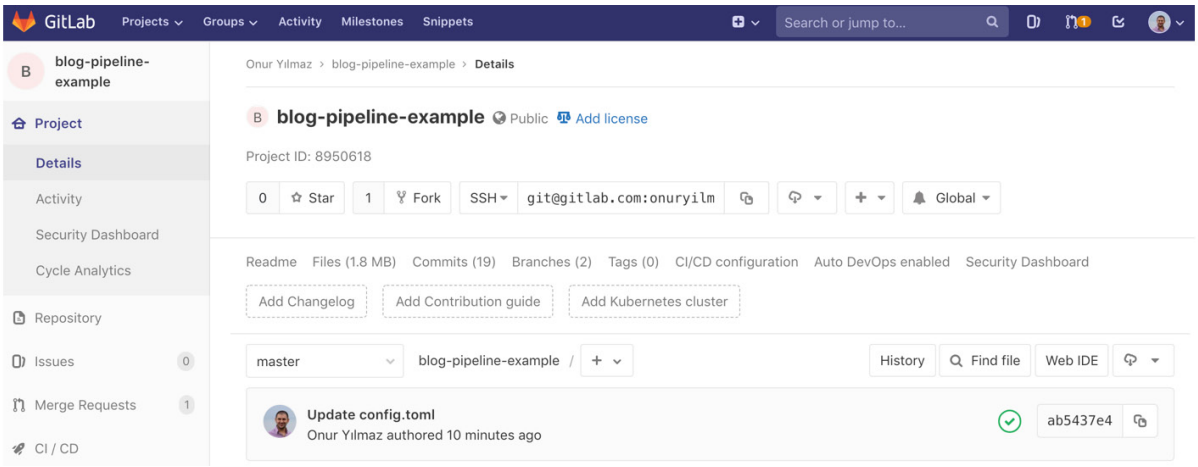

Figure 1.8: Forking the project to create a copy

2. Review the hierarchy of the files and their usages by running the tree command in the shell:

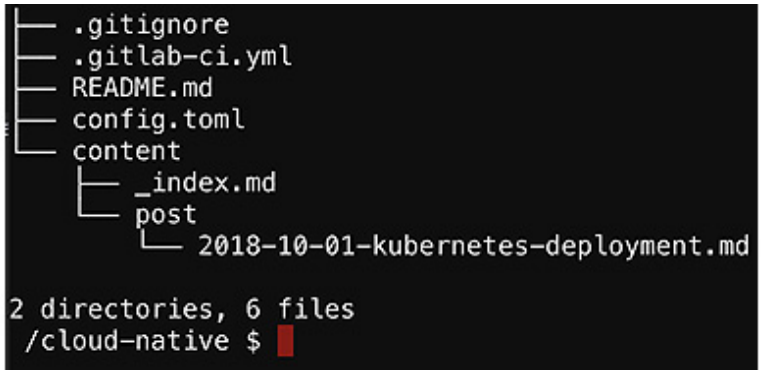

Figure 1.9: Tree view of the folder

As we can see, the following files appear on the tree: **`.gitignore`** is used for ignored files in the repository, while **`.gitlab-ci.yaml`** defines CI/CD pipeline on GitLab. The **`config.toml`** file defines the configuration for Hugo. The **`content`** folder is for the source content of the blog and contains another folder, **`post`**, and a **`_index.md`** file. The **`post`** folder contains another file called **`2018-10-01-kubernetes-deployment.md`**. Details on these are provide as follows. **`_index.md`** is a Markdown style for the index page, and **`2018-10-01-kubernetes-deployment.md`** is the only blog post live now.

3. Open the CI/CD pipeline defined in the **`.gitlab-ci.yaml`** file and check the following code:

```
image: registry.gitlab.com/pages/hugo:latest

stages:

  - validate

  - pages



validate:
```

```
  stage: validate

  script:

  - hugo

pages:

  stage: pages

  script:

  - mkdir -p themes/beautifulhugo && git clone
https://github.com/halogenica/beautifulhugo.git
themes/beautifulhugo

  - hugo --theme beautifulhugo

  only:

  - master

  artifacts:

    paths:

    - public
```

The **`image`**: block defines the Docker container image where the pipeline steps will be running, and the **`stages`**

block defines the sequential order of the jobs that will run. In this pipeline, first, **`validate`** will run and if it is successful, then **`pages`** will run.

The **`validate`** block defines the required testing before publishing the changes. In this block, there is only one command: **`hugo`**. This command verifies whether the contents are correct for creating a website.

The **`pages`** block is for generating the website with its template and finally publishing it. In the **`script`** section, first, the template is installed, and then the site is generated with the installed theme. However, **`pages`** has an **`only`** block with a **`master`**. This implies that only for the **`master`** branch will the website be updated. In other words, the pipeline could run for other branches for validation, but the actual site will only be deployed from the **`master`** branch.

4. Check the CI/CD pipelines from the left-hand side of the menu bar by clicking **Pipelines** under the **CI/CD** tab, as shown in the following screenshot:

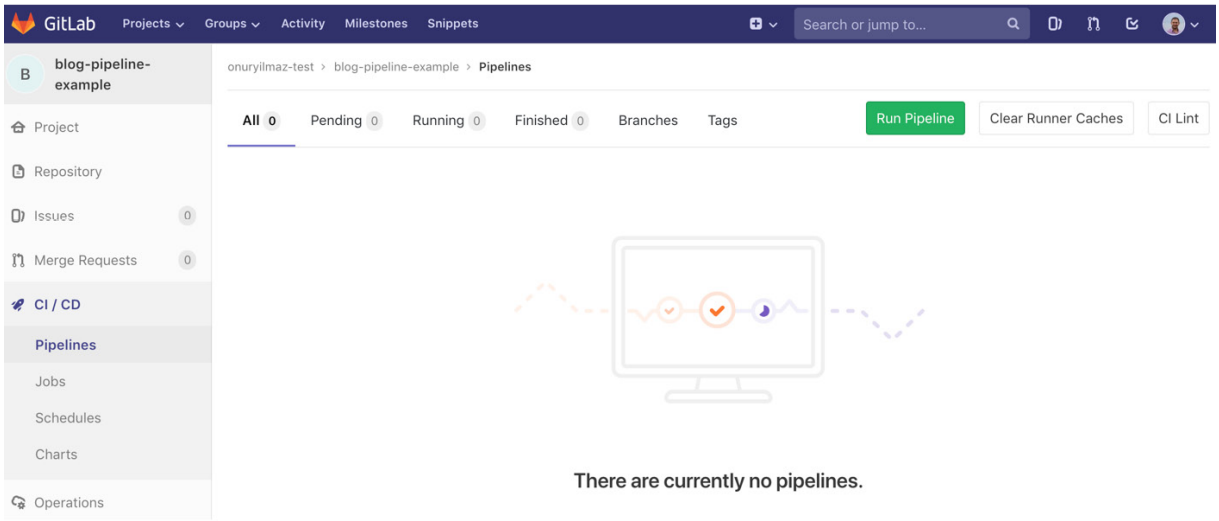

Figure 1.10: The CI/CD pipeline view

As expected, the page shows that there are no pipelines as we have not yet created them.

5. Click **Run Pipeline** on the top right-hand corner of the interface. It will redirect you to the following page; then, click **Create pipeline**:

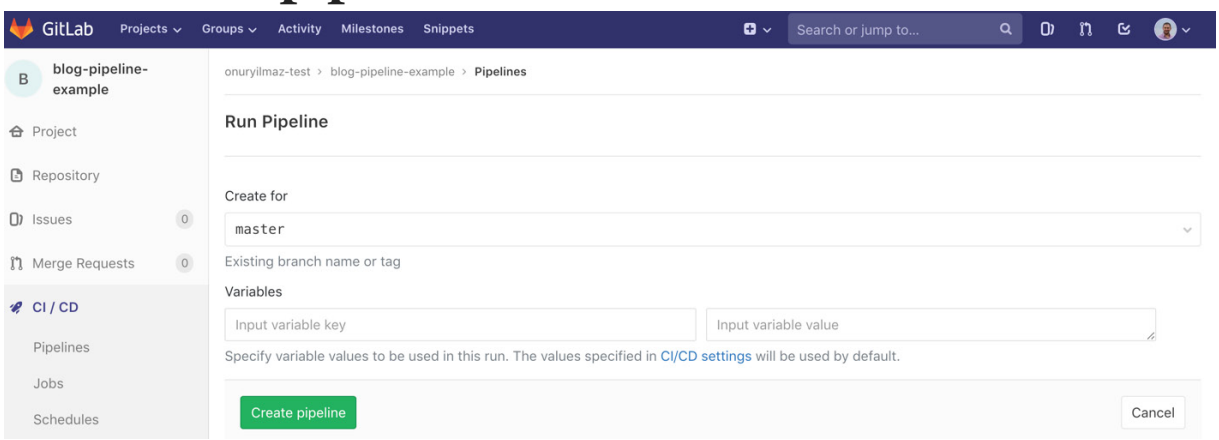

Figure 1.11: Creating a pipeline on GitLab

You will be able to view the running pipeline instance. With a successful run, it is expected that you will see three successful jobs for **Validate**, **Pages**, and **Deploy**, as shown in the following screenshot:

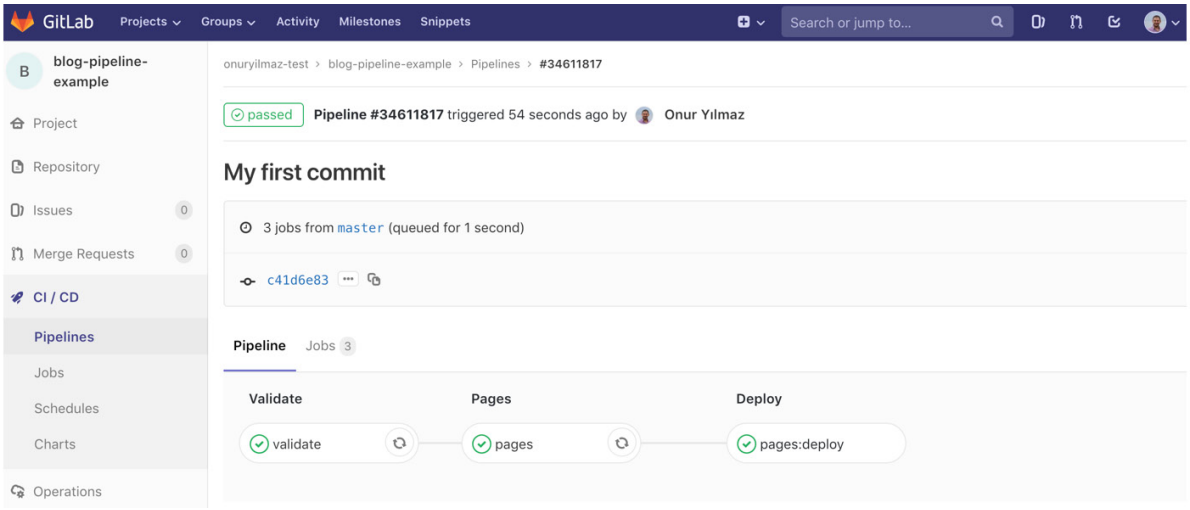

Figure 1.12: Successful Validate, Pages, and Deploy jobs

6. Click on the **pages** tab, as displayed in the preceding screenshot. You will obtain the following output log:

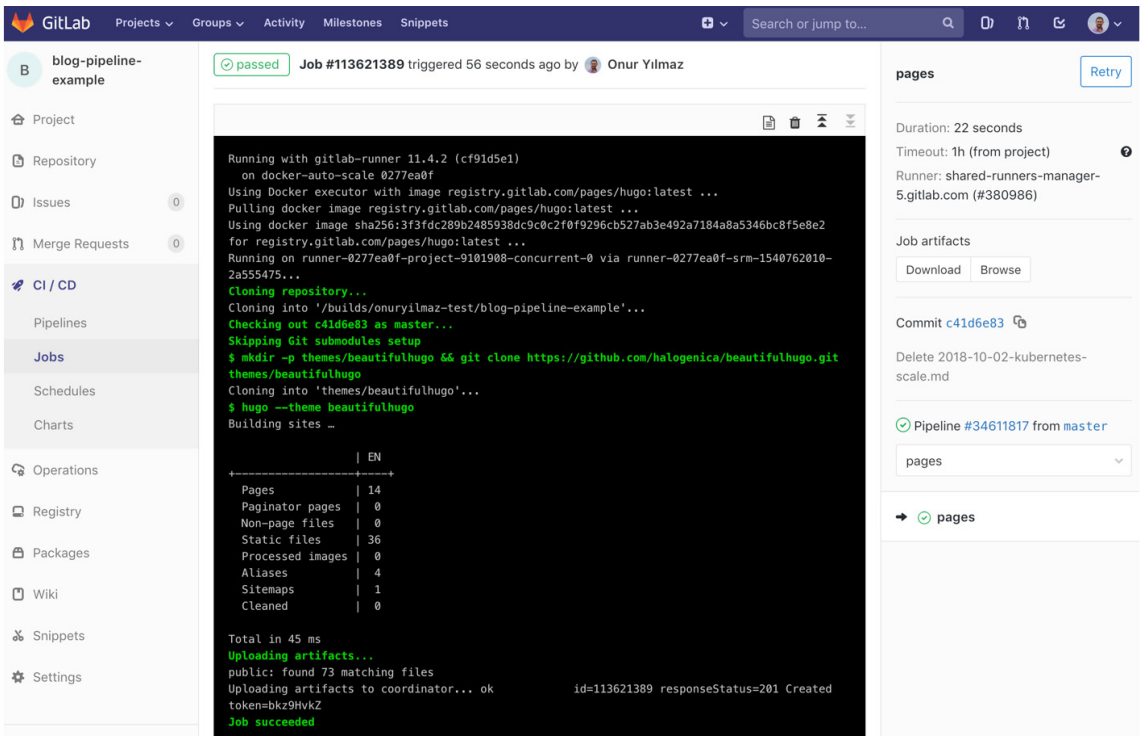

Figure 1.13: A log of the jobs run in containers

In the log screen, all of the lines until **`Cloning repository...`** show the preparation steps for the build environment. After that, the source code repository is cloned, and the **`beatifulhugo`** template is retrieved. This part is essential since it enables us to combine blog

and style at the build time. This approach makes it possible to change to another styling with a template in the future easily without source code change. Then, the HTML files are generated in the **`Building sites`** part, and finally, artifacts are uploaded to be served by **GitLab**.

7. Type the following URL in your browser window: **`https://<USERNAME>.gitlab.io/blog-pipeline-example/`**. The published website is shown as follows:

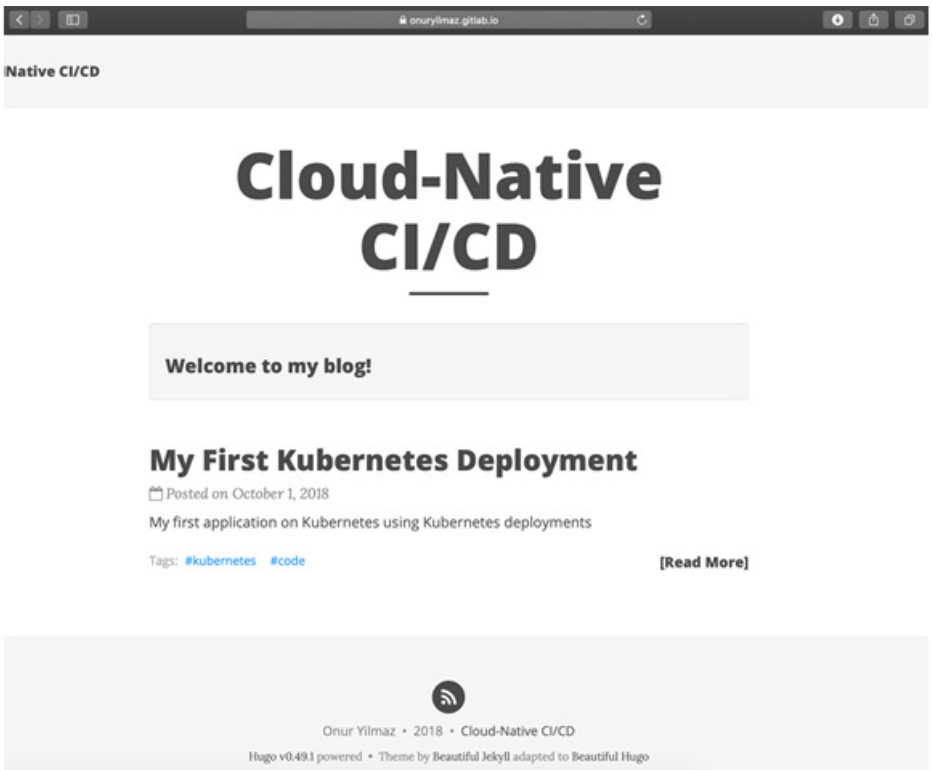

Figure 1.14: Screen shot of the live blog

Note

It could take up to 10 minutes for DNS resolution of the **`<USERNAME>.gitlab.io`** *address. If you see a* **`"404 - The page you're looking for could not be`**

found" *error, then please ensure that your address is correct and wait patiently until your blog works.*

8. Create another file with the name **2018-10-02-kubernetes-scale.md** under the **content/post** folder and type in the following code:

-

title: Scaling My Kubernetes Deployment

date: 2018-10-02

tags: ["kubernetes", "code"]

//[...]

NAME READY STATUS RESTARTS AGE IP NODE

kubernetes-bootcamp-5c69669756-9jhz9 1/1 Running 0 3s 172.18.0.7 minikube

kubernetes-bootcamp-5c69669756-lrjwz 1/1 Running 0 3s 172.18.0.5 minikube

kubernetes-bootcamp-5c69669756-slht6 1/1 Running 0 3s 172.18.0.6 minikube

```
   kubernetes-bootcamp-5c69669756-
t4pcs   1/1  Running   0        28s 172.18.0.4   minikube

"""
```

We are doing this as we expect the blog to be updated with new content thanks to the CI/CD pipeline we activated with the `.gitlab-ci.yaml` file.

Note

The file and the complete code is available under the new-post branch of this project: https://bit.ly/2rz8O3Y.

9. Click on **Pipelines** under the **CI/CD** tab to check whether there are further instances of the pipeline. The CI/CD pipeline automatically runs and creates a new deployment as soon as a new post is added:

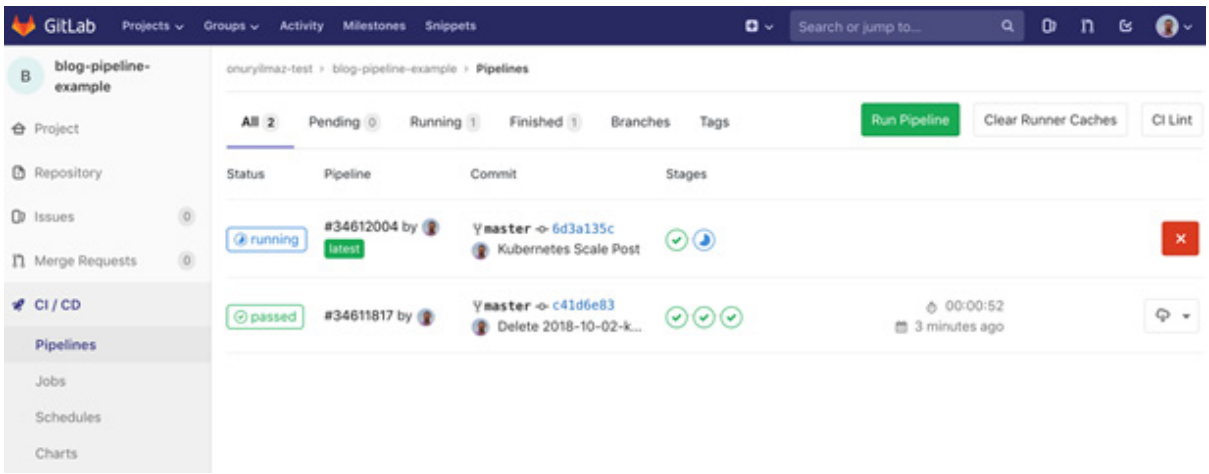

Figure 1.15: The pipeline runs whenever there is a new commit

10. Type `https://<USERNAME>.gitlab.io/blog-pipeline-example/` and check whether the website is

updated when the pipeline is finished:

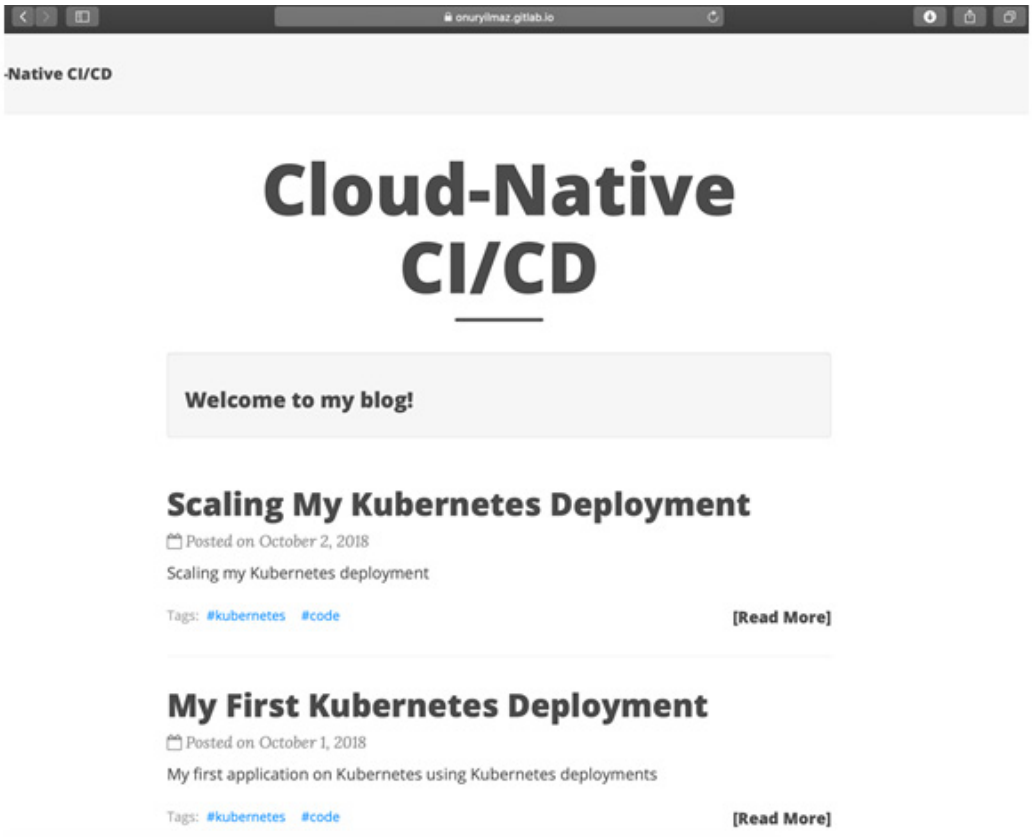

Figure 1.16: The website is updated with the new post automatically

As you can see, compared to the previous output (as listed in step 7), the website has now been updated.

Thus, in this exercise, we created a blog where the contents were written in a repository, and we observed that the site was automatically generated and published by CI/CD pipelines. We will now summarize this chapter.

Summary

In this chapter, we first described the conventional method of software development and established its limitations.

Specifically, we described how conventional methods failed to encourage collaboration between development and operations, thus ultimately resulting in the loss of engineer hours and money. Then, we discussed the motivation for the origin of the DevOps culture shift. We expanded the discussion by listing DevOps best practices and introduced the DevOps toolchain.

We then progressed to introduce cloud-native architectures and described how it compliments DevOps in bringing about a paradigm shift in software development. Also presented in this chapter was a set of guidelines that will help you choose the best CI/CD tools to implement two critical cloud-native DevOps patterns, namely continuous integration and continuous delivery/deployment, for enhanced collaboration. Finally, we ended this chapter by creating and running a pipeline for a blog application on **GitLab**.

In the next chapter, we will be describing the fundamentals of continuous integration for the cloud-native architecture and introduce container technology. Additionally, we will be identifying and running several levels of testing for microservices.

Cloud-Native Continuous Integration

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the fundamentals of continuous integration for cloud-native architecture
- Identify different levels of testing for microservices
- Run each level of testing in a cloud-native way
- Describe critical points for building microservice containers
- Design a complete continuous integration pipeline for a cloud-native microservice

This chapter presents the introduction to cloud-native continuous integration and container technology along with guidelines to create a complete pipeline for a cloud-native microservice.

Introduction

In this book, we first discussed the DevOps culture shift, with its best practices and toolchain compared to conventional software development. Then, cloud-native architecture design and how DevOps practices with cloud-native applications have created a paradigm shift was presented. In this chapter, the first DevOps practice for cloud-native applications, namely continuous integration, will be covered. First, we will explain container technology, along with a brief history of it. Following that, cloud-native application testing and the building of containers will be discussed. Finally, a checklist for a CI pipeline for a cloud-native application will be presented.

CLOUD-NATIVE CONTINUOUS INTEGRATION

CI is based on the idea of integrating changes from different sources as soon as possible in a continuous way. Integration consists of all testing stages, from unit to integration tests, including reviewing the code changes and building the executables successfully. The main idea of CI is to find bugs and conflicts as quickly as possible and solving these early in the softwares life cycle. There are two main benefits of CI that are interdependent: resolving conflicts early and increasing the quality.

The CI mindset focuses on the early consideration to integrate instead of waiting for a big bang of hundreds of code changes. Developers could find conflicts and solve inconsistencies early when the effects are not significant. When the conflicts are resolved, all parties can work on the approved and tested, namely the green codebase, with trust. It increases the quality of software that is delivered by increasing the collaboration and clear understanding of requirements. For instance, let's

think about a group of developers working on a shopping website. The backend developers change a field of customer data, run their unit tests locally, and finally push their code. When new code is pushed, the CI pipeline should automatically run integration tests, combining frontend and backend, and should finally turn **red**. This automatic integration and alerting the teams allows you to find the inconsistencies earlier and solve them before coming face to face with a huge problem. As a result, all stakeholders work on better quality software with automation and collaboration.

As cloud-native architecture changed the way of developing software, it also changed the CI methods. There are two main drivers of cloud-native architecture in CI: **microservices** and **containers**.

First, applications are designed as loosely coupled services that exist independent of each other, namely microservices. Each service focuses only on its respective functionalities and does not directly depend on the others. These microservices are expected to be tested and built separately, and finally integrated to achieve broad business functions. Secondly, cloud-native applications are packaged and delivered as containers. Each container consists of minimum requirements, such as the operating system and dependency libraries of the service to be as lightweight as possible. Therefore, CI pipelines are expected to create the minimum required lightweight containers as deliverables.

The goal of cloud-native continuous integration is to have an automated, repeatable pipeline where the microservices are tested and built as lightweight containers.

In the following section, we will start with container technology as being the building block of cloud-native microservice applications.

Container Technology

Container technology has been around for nearly two decades and has been launched by different companies in several domains. **Oracle Solaris** with **Zones** and the **FreeBSD** operating system with **Jails** are the first prominent operating system-level virtualization. Open source Linux containers that focus on sharing the operating system and kernel to create lightweight containerization are the most popular ones today. Open source containers are scalable, robust, and are self-proven, since every significant Google functionality such as Gmail or Google Maps has been running in containers for a very long time.

Docker, which is a Linux container, changed the path of containerization by making it easier and safer to develop and deploy. Collaboration with large enterprise companies that have been using containers for a long time, such as Google, Red Hat, and Canonical, increased the popularity of Docker. In addition, the open source container runtime of Docker was donated to The Linux Foundation's Open Container Project for creating an industry standard. With the standardization and large-scale use of containers, new implementations of Linux containers are emerging, such as **`rkt`** by **CoreOs** and **`LXD`** by **Canonical**. However, Docker is still essentially the most prominent in both the open source world and enterprise.

In the following section, the technology behind the

standardized Linux containers will be discussed, since they enable the cloud-native applications of the future.

LINUX CONTAINERS

Containers in Linux are merely three fundamental features of the operating system working coherently. These three features are **namespaces**, **control groups** (**cgroups**), and layered **filesystems,** which create isolation and apply limitations for each container, as shown in the following diagram:

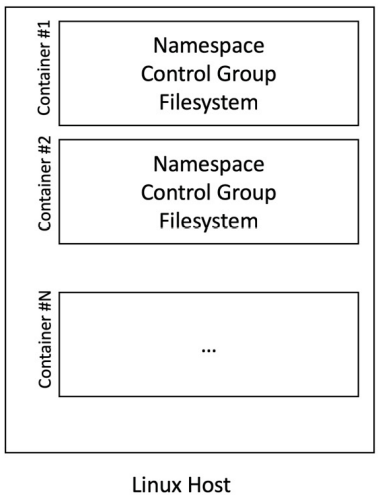

Figure 2.1: Multiple containers in a Linux host

Namespaces: Namespaces isolate each application from the host and other applications by creating separate environments. Namespaces for the following resources are assigned to the containers, and the access of the container is limited only by the corresponding namespace:

- **`pid`** (process ID): Process isolation
- **`net`** (network): Network interfaces
- **`ipc`** (Inter-Process Communication): Access to IPC resources

- **`mnt`** (mount): Filesystem mount points
- **`uts`** (Unix timesharing system): Isolating kernel and version identifiers

Namespace isolation creates a special environment so that the **`pid`** of the main application in the container becomes "1". Running a **`ps`** command shows **`pid`** 1 and its children processes only. With the same approach, the application could connect to any network port without any conflict in other applications in other containers due to **`net`** namespace isolation. For instance, you can start multiple MySQL containers, each using the 3306 port in their own container without any conflict. Mounting volumes to the containers or making IPC calls are also isolated in each Container such as other namespaces, to create a completely private environment.

Control Groups: In Linux, **`cgroups`** are used to limit applications for a specific set of resources, such as memory or processing power. Containerization uses **`cgroups`** to share resources among containers by enforcing limits. While namespaces isolate containers, **`cgroups`** define limitations for containers to share resources.

Layered Filesystems: Layered filesystems consist of reusable layers that are stacked on top of one another to form a root filesystem. It is the primary technology that enables containers to be lightweight. Namespaces and **`cgroups`** surround the containers to be isolated and limited, whereas a layered filesystem consists of what is shared and packaged inside the containers.

Until now, we have introduced cloud-native CI and container technology. We will test and build microservices and utilize the features of containers with the best practices in the following sections.

Testing Cloud-Native Applications

Testing cloud-native microservices is critical for creating robust, reliable, and scalable applications. It is crucial to have an automated CI pipeline that tests the main branch of the application and creates a checkpoint for pull requests from other branches. There are well-defined and established levels of testing in the industry; however, in this section, these levels will be discussed for a cloud-native microservice.

Before diving into the levels of testing, let's consider creating a sample API. This API will be created with the perspective of organizing, retrieving, and analyzing information on books in a library. A sample **`book-server`** was designed with cloud-native microservice architecture in mind and will be used for defining different levels of test in this section.

Note

The source code of the book-server API can be found here: https://gitlab.com/TrainingByPackt/book-server.

Keeping the microservice architecture in perspective, we need to design and implement self-sufficient services for accomplishing business needs. Hence, we will create a microservice that works as a REST API to interact with the book information. We do not need to consider how book

information is stored in a database, since it should be a separate service in our architecture. In other words, we need to create a REST API server that works with any SQL capable database and design it to be compatible with other types of databases in the future.

Before diving into the details of **`book-server`**, let's take a look at the structure of the repository by using the tree command. With the following command, all of the files and folders are listed, except for the vendor folder, where the source code of Go dependencies are kept:

tree -I vendor –U

You can view the files and folder in the following output:

Figure 2.2: Tree view of the folders

By following the best practices in Go development, the book-server is structured as follows:

- **`The cmd`** folder includes main.go, which creates the executable for the book-server

- **`The docker`** folder includes Dockerfiles that will be used for different testing levels such as static code check, unit test, smoke test, and integration test
- **`Dockerfile`** is the container definition that's used for building book-server
- **`Makefile`** contains the commands to test and build the repository
- **`README.md`** includes documentation on the repository
- **`The pkg`** folder consists of the source code of the book-server
- **`pkg/books`** defines the database and book interfaces for extending book-server
- **`pkg/commons`** defines the option types that are used within book-server
- **`pkg/server`** consists of the HTTP REST API code and related tests

Important points related to the testing and building of book-server are illustrated with the following code explanations:

For our API, **`book-server`** is a Go REST API microservice that connects to a database and serves information about books. In the **`cmd/main.go`** file, it can be seen that the service only works with three parameters:

- Log level
- HTTP port
- Database address

In the following **`init`** function, in **`main.go`**, these three parameters are defined as command-line arguments as log-level, port, and db:

```
func init() {

pflag.StringVar(&options.ServerPort, "port", "8080", "Server port for listening REST calls")

pflag.StringVar(&options.DatabaseAddress, "db", "", "Database instance")

pflag.StringVar(&options.LogLevel, "log-level", "info", "Log level, options are panic, fatal, error, warning, info and debug")

}
```

It is expected that you should have different levels of logging in microservices so that you can debug services running in production better. In addition, ports and database addresses should be configured on the fly, since these should be the concerns of the users, not developers.

In the **`pkg/books/model.go`** file, **`Book`** is defined, and an interface for book database, namely **`BookDatabase`**, is provided. It is crucial for microservices to work with interfaces instead of implementations, since interfaces enable plug-and-play capability and create an open architecture. You can see how **`book`** and **`BookDatabase`** are defined in the following snippet:

```
type Book struct {

ISBN string
```

```
Title string

Author string

}

type BookDatabase interface {

GetBooks() ([]Book, error)

Initialize() error

}
```

Note

The code files for this section can be found here: https://bit.ly/2S92tbr.

In the **`pkg/books/database.go`** file, a SQL capable **`BookDatabase`** implementation is developed as **`SQLBookDatabase`**. This implementation enables the book-server to work with any SQL capable database. The **`Initialize`** and **`GetBooks`** methods could be checked for how SQL primitives are utilized to interact with the database. In the following code fragment, the **`GetBooks`** and **`Initialize`** implementations are included, along with their SQL usages:

```
func (sbd SQLBookDatabase) GetBooks() ([]Book, error) {

  books := make([]Book, 0)

  rows, err := sbd.db.Query('SELECT * FROM books')
```

```
  //[...]

  return books, nil

}

func (sbd SQLBookDatabase) Initialize() error {

  var schema = 'CREATE TABLE books (isbn text, title text,
author text);'

//[...]

return nil

}
```

Finally, in the **`server/server.go`** file, an HTTP REST API server is defined and connected to a port for serving incoming requests. Basically, this server implementation interacts with the **`BookDatabase`** interface and returns the responses according to HTTP results.

In the following fragment of the **`Start`** function in **`server.go`**, endpoints are defined and then the server starts to listen on the port for incoming requests:

```
func (r *REST) Start() {

  //[...]

   r.router.GET("/ping", r.pingHandler)

   r.router.GET("/v1/init", r.initBooks)
```

```
    r.router.GET("/v1/books", r.booksHandler)

    r.server = &http.Server{Addr: ":" + r.port, Handler:
r.router}

     //[...]

     err := r.server.ListenAndServe()

      //[...]

}
```

Note

The complete code can be found here: https://bit.ly/2Cm9Mag.

In the preceding section, a cloud-native microservice application, namely book-server, was presented, along with its important features. In the next section, we will begin with static code analysis so that we can test this application comprehensively.

STATIC CODE ANALYSIS

Reading and finding flaws in code is cumbersome and requires many engineering hours. Therefore, it is beneficial to use automated code analysis tools that analyze the code and find potential problems. It is a crucial step and should be located in the very first stages of the CI pipeline. Static code analysis is essential because correctly working code with the wrong style will cause more damage than non-functional code.

It is beneficial for all levels of developers and quality teams to follow standard guidelines in the programming languages and create their styles and templates only if necessary. There are many static code analyzers that are available one the market as services or open source:

- Pylint for Python
- FindBugs for Java
- SonarQube for multiple languages and custom integrations
- The IBM Security AppScan Standard for security checks and data breaches
- JSHint for JavaScript

However, while choosing a static code analyzer for a cloud-native microservice, the following three points should be considered:

- **The best tool for the language**: It is common to develop microservices in different programming languages; therefore, you should select the best static code analyzer for the language rather than employing one-size-fits-all solutions.
- **Scalability of the analyzer**: Similar to cloud-native applications, tools in software development should also be scalable. Therefore, select only those analyzers that can run in containers.
- **Configurability**: Static code analyzers are configured to run and find the most widely accepted errors and flaws in the source code. However, the analyzer should also be

configured to different levels of checks, skipping some checks or adding some more rules to check.

EXERCISE 2: PERFORMING STATIC CODE ANALYSIS IN CONTAINERS

In this exercise, a static code analyzer for the book-server application will be run in a Docker container. Static code analysis will check for the source code of book-server and list the problematic cases, such as not checking the error returned by functions in Go. To complete this exercise, the following steps have to be executed:

Note

All tests and build steps are executed for the book-server application in the root folder. The source code of the book-server is available on GitLab: https://gitlab.com/TrainingByPackt/book-server.The code file for this exercise can be found here: https://bit.ly/2EtBoNy.

1. Open the **`docker/Dockerfile.static-code-check`** file from the GitlLab interface and check the container definition for the static code analysis:

   FROM golangci/golangci-lint

   ADD . /go/src/gitlab.com/onuryilmaz/book-server

   WORKDIR /go/src/gitlab.com/onuryilmaz/book-server

   RUN golangci-lint run ./...

2. Build the container in the root directory of book-server by running the following code:

   docker build --rm -f docker/Dockerfile.static-code-check .

   In the preceding file, the **`golangci/golangci-lint`** image is used as the static code analysis environment and the **`book-server`** code is copied. Finally, **`golangci-lint`** is run for all folders to find flaws in the source code.

   The following output is obtained once the preceding code is run with no errors, with a **`Successfully built`** message at the end:

```
 /cloud-native $ docker build --rm -f docker/Dockerfile.static-code-check .
Sending build context to Docker daemon  7.035MB
Step 1/4 : FROM  golangci/golangci-lint
 ---> 6ca05230d41a
Step 2/4 : ADD . /go/src/gitlab.com/onuryilmaz/book-server
 ---> 0d7003b5d02e
Step 3/4 : WORKDIR /go/src/gitlab.com/onuryilmaz/book-server
 ---> Running in 53ec1909b392
Removing intermediate container 53ec1909b392
 ---> f84490802a18
Step 4/4 : RUN golangci-lint run ./...
 ---> Running in 451d1e481c67
Removing intermediate container 451d1e481c67
 ---> a0f222bf4bf7
Successfully built a0f222bf4bf7
 /cloud-native $
```

   Figure 2.3: Output of the static code analysis

   It is expected that you see no errors and that you have successfully built the container, since the source code has no flaws.

3. Change the **`Initialize`** function in **`pkg/books/database.go`** as follows by removing error checks in the SQL statements:

   func (sbd SQLBookDatabase) Initialize() error {

   var schema = 'CREATE TABLE books (isbn text, title text, author text);'

```
sbd.db.Exec(schema)

var firstBooks = 'INSERT INTO books ...'

sbd.db.Exec(firstBooks)

return nil

}
```

With the modified **`Initialize`** function, the responses of the **`sbd.db.Exec`** methods are not checked. If these executions fail with some errors, these return values are not controlled and not sent back to caller functions. It is a bad practice and a common mistake in programming that's mostly caused by the assumption that the code will always run successfully.

4. Run the following command, as we did in step 2:

```
docker build --rm -f docker/Dockerfile.static-code-check .
```

Since we had modified the code is step 3, it is expected that we should see a failure as a result of this command, as shown in the following screenshot:

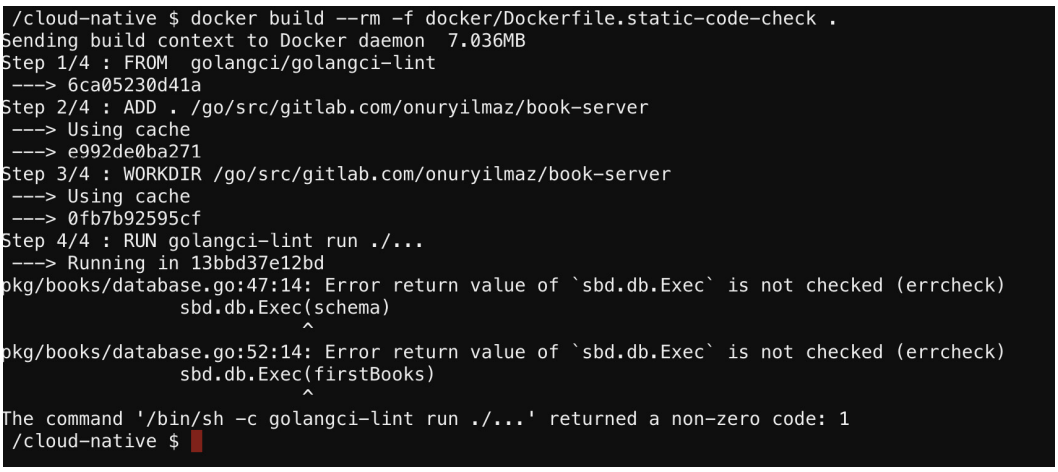

Figure 2.4: Output obtained with the modified Initialize function

As we can see, **`errcheck`** errors are expected, since we are not checking for the errors during SQL executions.

5. Revert the code for the **`Initialize`** function to the original, with error checks where static code analysis successfully completed; otherwise, the static code analysis step will always fail in the pipeline and the further steps will never run.

After automatic analysis of the source code and finding the potential problems, we will start with the building block of testing—unit testing.

UNIT TESTING

Unit testing focuses on testing every small unit of the software in isolation and making sure that it runs correctly. The mainstream approach for unit testing is to test each function with the mocks provided for the dependent libraries and resources. Since microservices are already narrow-scoped to single business functions, it could be difficult to find smaller units; however, it is suggested that you cover all functions that are critical for business operations. In addition, all external dependencies should be isolated in the unit testing of microservices. It is also common for developers to run unit tests locally before pushing the code; therefore, unit tests should be designed to run on both the development environment and the CI pipeline. Unit tests in microservices ensure that the underlying foundation of higher-level software systems are running as designed; therefore, they are critical to minimizing integration efforts.

In the following exercise, all functional units of book-server will be tested. Since book-server is a REST API microservice, the tests will focus on testing API endpoints in an isolated environment.

EXERCISE 3: PERFORMING UNIT TESTING FOR MICROSERVICES

In this exercise, unit tests for **`book-server`** microservice are run, and the results are inspected. The core functionality of **`book-server`** is serving REST queries over its defined endpoints. Therefore, in unit tests, the endpoints of the server will be checked without actually connecting to a database:

1. Open the **`pkg/server/server_test.go`** file and check the unit test that was developed as **`TestBookServer`**:

   ```
   func TestBookServer(t *testing.T) {

   //[...]

   options.Books = MockBookDatabase{}

   //[...]

   RESTServer := NewREST(options)

   RESTServer.Start()

   Convey("Start book server with mocked database", t,
   func() {

   Convey("Ping server", func() {

   //[...]
   ```

```
Convey("Init books database", func() {

//[...]

Convey("Get books", func() {...}
```

Note

The code files for this exercise can be found here: https://bit.ly/2CnhctX.

In the preceding **`TestBookServer`** function fragment, there are two critical points to mention. The first one is that all three endpoints are tested in **`Convey`** blocks, since this is a unit-testing stage. This is important, since unit tests should cover all unit functionalities of the services. The second point is that a **`MockBookDatabase`**, which is an implementation of the **`BookDatabase`** interface, is used. It is the fundamental characteristic of unit tests that each function should be tested in isolation. Therefore, the server functionality of book-server is tested with a mock database with a dummy implementation. In the following code, **`MockBookDatabase`** is provided:

```
type MockBookDatabase struct{}

func (mbd MockBookDatabase) GetBooks()
([]books.Book, error) {

b := make([]books.Book, 0)

b = append(b, books.Book{ISBN: "ISBN", Title: "Title",
Author: "Author"})
```

```
return b, nil

}

func (mbd MockBookDatabase) Initialize() error {

return nil

}
```

`MockBookDatabase` is merely implemented to send expected results so that the server part can be tested. For instance, the **`GetBooks`** function just returns a dummy book without getting any data from a database or any other source. With the same approach, the I**`nitialize`** function just returns successfully, with no errors in the server code.

2. Open the **`docker/Dockerfile.unit-test`** file and check the container definition for the unit tests:

```
FROM golang:1.10

ADD . /go/src/gitlab.com/onuryilmaz/book-server

WORKDIR /go/src/gitlab.com/onuryilmaz/book-server

RUN go test ./... -v -tags=unit
```

In this container definition, **`golang:1.10`** is used as the testing environment, and all of the source code is copied. At the end of this container, **`go test`** is run for all the folders to run the tests with the **`unit`** tag.

3. Run the unit tests in the root folder of **`book-server`** and

check the results by running the following command:

```
docker build --rm -f docker/Dockerfile.unit-test .
```

You will obtain the following output:

```
 /cloud-native $ docker build --rm -f docker/Dockerfile.unit-test .
Sending build context to Docker daemon  7.084MB
Step 1/4 : FROM golang:1.10
 ---> 0a19f4d16598
Step 2/4 : ADD . /go/src/gitlab.com/onuryilmaz/book-server
 ---> 4de62268fe26
Step 3/4 : WORKDIR /go/src/gitlab.com/onuryilmaz/book-server
 ---> Running in 32484f38a3f5
Removing intermediate container 32484f38a3f5
 ---> d4ab9e992de5
Step 4/4 : RUN go test ./... -v -tags=unit
 ---> Running in 9a550d0547ec
?       gitlab.com/onuryilmaz/book-server/cmd   [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/books     [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/commons   [no test files]
=== RUN   TestBookServer
time="2018-11-19T11:50:16Z" level=info msg="Starting REST server..."
time="2018-11-19T11:50:16Z" level=info msg="REST server connecting to port 36647"

  Start book server with mocked database ✔
    Ping server ✔✔✔
    Init books database ✔✔✔
    Get books ✔✔


9 total assertions

--- PASS: TestBookServer (3.01s)
PASS
ok      gitlab.com/onuryilmaz/book-server/pkg/server    3.020s
Removing intermediate container 9a550d0547ec
 ---> c617783a685c
Successfully built c617783a685c
 /cloud-native $
```

Figure 2.5: Output of the unit test

The results show that a total of nine assertions were completed, which shows that the REST API server is working as expected in isolation. After the unit testing, it is shown that the core functionality of book-server, namely the REST API server, works as expected.

In the following section, a checkpoint will be tested for **`book-server`** to verify whether it is working when it is integrated with other services.

SMOKE TESTING

Smoke testing creates a checkpoint for continuing with further integration tests or not. These are necessary inspections that should be checked after the integration of multiple components, such as checking whether the database is connected to the backend, or whether the frontend can connect to the backend services. If smoke tests fail, in other words, there are "smokes" after the integration, there is no need to run integration tests. Although smoke testing seems like a superficial testing step, it is beneficial to avoid running integration tests that will most likely fail, with hundreds of alerts and notifications being provided.

In the following exercise, the **`book-server`** application will be tested by smoke testing as a checkpoint for further integration tests.

EXERCISE 4: PERFORMING SMOKE TESTS FOR MICROSERVICES

In this exercise, smoke tests for the **`book-server`** microservice are run, and the results are inspected. The **`book-server`** microservice is designed to connect database services during integration. Therefore, the smoke-testing basic functionality of the **`book-server`** will be checked after connecting to a database instance:

1. Open the **`pkg/server/server_smoke_test.go`** file and check the smoke test that was developed as **`TestBookServerSmoke`**:

   func TestBookServerSmoke(t *testing.T) {

   //[...]

```
db, err := books.NewSQLBookDatabase(databaseAddress)

options.Books = db

RESTServer := NewREST(options)

RESTServer.Start()

Convey("Start book server with database", t, func() {

//[...]

Convey("Ping server", func() {

//[...]

}
```

In the preceding code, there are two critical points to mention. First, only the ping endpoint of the REST API server is tested as a checkpoint. Second, a real database instance is connected to the API server. This approach tests the API server to check whether it is still working after connecting to a database server.

2. Open the **`docker/Dockerfile.smoke-test`** file and review the container definition for smoke tests:

```
FROM golang:1.10

ADD . /go/src/gitlab.com/onuryilmaz/book-server

WORKDIR /go/src/gitlab.com/onuryilmaz/book-server

ENV DATABASE ""
```

RUN go test ./... -v -tags=smoke -db=$DATABASE

In this container definition, **`golang:1.10`** is used as the testing environment, and all source code is copied. At the end of this container, **`go test`** is run for all the folders to run the tests with the **`smoke`** tag.

3. Build the smoke test Docker container by running the following command:

   docker build -f docker/Dockerfile.smoke-test -t onuryilmaz/book-server/smoke-test:latest .

   This command will build a container image for **`smoke-test`**. Smoke tests check the microservices after connecting to other services. We need an actual Docker container to run, and so we will start by building it.

4. Start and wait for the database by running the following command:

   docker run -d -p 5432:5432 --name postgres postgres

   docker run --rm --link postgres:postgres gesellix/wait-for postgres:5432

   The first command starts a **`postgres`** database container with the name of **`postgres`** and publishes the **`5432`** port to localhost. With the **`-d`** flag, the **`postgres`** container runs in the background like a **`daemon`** service. The second command runs a **`wait-for`** container to waiting until the **`5432`** port of the **`postgres`** container is reachable. With the successful run of these two commands, no output is expected in the command line.

5. Run the smoke test container by connecting to the database that was initiated in step 4:

   docker run -e DATABASE="postgresql://postgres:postgres@postgres:5432/post sslmode=disable" --link postgres onuryilmaz/book-server/smoke-test:latest

   You will obtain the following output:

```
docker run -e DATABASE="postgresql://postgres:postgres@postgres:5432/postgres?sslmode=disable"
 --link postgres  onuryilmaz/book-server/smoke-test:latest
?       gitlab.com/onuryilmaz/book-server/cmd   [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/books     [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/commons   [no test files]
=== RUN   TestBookServerSmoke
time="2018-11-19T11:53:00Z" level=info msg="Starting REST server..."
time="2018-11-19T11:53:00Z" level=info msg="REST server connecting to port 42389"

  Start book server with database ✔✔
    Ping server ✔✔


4 total assertions

--- PASS: TestBookServerSmoke (1.01s)
PASS
ok      gitlab.com/onuryilmaz/book-server/pkg/server    1.013s
docker stop postgres
postgres
docker rm postgres
postgres
 /cloud-native $
```

Figure 2.6: Output of the smoke test

   The results show that a total of four assertions were completed, which shows that the REST API server is further testable after connecting to the database. As expected, the smoke tests work as checkpoints for further integration tests.

6. Stop and remove the database that we started in step 4 by running the following command:

   docker stop postgres

   docker rm postgres

With this exercise, a smoke test is used to check whether there

are any smokes after connecting multiple microservices. We can see that book-server is still up when it is connected to a real database, so the comprehensive integration tests in the next section can be started.

INTEGRATION TESTING

Integration testing focuses on ensuring that multiple components work together as designed. These tests are usually the most time-consuming tests, where multiple microservices are connected and business functionalities are checked. For cloud-native applications, integration tests are run as each microservice in a separate container that connects over their APIs, like in a production environment. In the unit tests, dependent libraries were replaced with mock implementations; however, in this step, the actual libraries are used so that the data flow between components is realized. Integration tests, along with the previous stages, help to identify inconsistencies between components, and they should be run in CI pipelines automatically to find bugs and problems earlier.

EXERCISE 5: PERFORMING INTEGRATION TESTING FOR MICROSERVICES

In this exercise, integration tests for the **`book-server`** microservice are run, and the results are inspected. The **`book-server`** is a generic microservice that can connect any SQL capable database service and serve as a REST API. Therefore, it is critical in integration testing that all of the functionality of **`book-server`** is working as expected against all supported database services. To complete this exercise, the

following steps have to be executed:

1. Open the **`pkg/server/server_integration_test.go`** file and review the integration test that was developed as **`TestBookServerIntegration`**:

   func TestBookServerIntegration(t *testing.T) {

   ...

   db, err := books.NewSQLBookDatabase(databaseAddress)

   options.Books = db

   RESTServer := NewREST(options)

   RESTServer.Start()

   Convey("Start book server with database", t, func() {

   ...

   Convey("Ping server", func() {

   ...

   Convey("Init books database", func() {

   ...

   Convey("Get books", func() {

   ...}

   In this fragment of the function, two critical points related

to integration testing are covered. The first one is that all functionalities of the REST API server are tested. Second, a real database instance is connected to the API server. This approach tests whether the API server is working when connected to a database server.

2. Open the **`docker/Dockerfile.integration-test`** file and check the container definition for the integration tests:

   FROM golang:1.10

   ADD . /go/src/gitlab.com/onuryilmaz/book-server

   WORKDIR /go/src/gitlab.com/onuryilmaz/book-server

   ENV DATABASE ""

   RUN go test ./... -v -tags= integration -db=$DATABASE

   Similar to the previous testing steps, in this container definition, **`golang:1.10`** is used as a testing environment and all source code is copied. At the end of this container, **`go test`** is run for all the folders so that it can run the tests with the **`integration`** tag.

3. Build the Docker container for the integration test by running the following command:

   docker build -f docker/Dockerfile.integration-test -t onuryilmaz/book-server/integration-test:latest .

   This command will build a container image for the integration tests. The **`book-server`** service is designed to work with any SQL capable database. Therefore,

integration tests are expected to demonstrate this feature. Currently, **`book-server`** has been enabled with MySQL, Microsoft SQL Server, and PostgreSQL; therefore, the integration tests will run against each SQL server separately.

4. Start and wait for the MySQL database by running the following command:

   ```
   docker run -d -p 3306:3306 -e
   MYSQL_ROOT_PASSWORD=password -e
   MYSQL_DATABASE=default --name mysql mysql

   docker run --rm --link mysql:mysql gesellix/wait-for
   mysql:3306
   ```

   Similar to the smoke test exercise, the first command starts a **`mysql`** database container with the name of **`mysql`** and publishes the **`3306`** port to localhost. The second command runs a wait-for container to wait until the **`3306`** port of the **`mysql`** container is reachable. With the successful run of these two commands, no output is expected in the command line.

5. Run the Docker container for the integration test by connecting to the database that we started in the previous step by running the following command:

   ```
   docker run -e
   "DATABASE=mysql://root:password@mysql:3306/default"
   --link mysql onuryilmaz/book-server/integration-
   test:latest
   ```

   You will obtain the following output:

```
 /cloud-native $ docker run -e "DATABASE=mysql://root:password@mysql:3306/default" --link mysq
l onuryilmaz/book-server/integration-test:latest
?       gitlab.com/onuryilmaz/book-server/cmd    [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/books      [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/commons    [no test files]
=== RUN   TestBookServerIntegration
time="2018-11-19T15:37:35Z" level=info msg="Starting REST server..."
time="2018-11-19T15:37:35Z" level=info msg="REST server connecting to port 35497"

  Start and check RESTServer ✔✔
    Ping server ✔✔✔✔
    Init books database ✔✔✔✔
    Get books ✔✔


12 total assertions

--- PASS: TestBookServerIntegration (3.16s)
PASS
ok      gitlab.com/onuryilmaz/book-server/pkg/server    3.169s
 /cloud-native $
```

Figure 2.7: Output of the integration test for MySQL database

The output of this run indicates that 12 assertions were successfully completed, so the book-server passes integration tests with MySQL.

6. Stop and remove the MySQL database by running the following command:

   docker stop mysql

   docker rm mysql

7. Start and wait for the PostgreSQL database by running the following command:

   docker run -d -p 5432:5432 --name postgres postgres

   docker run --rm --link postgres:postgres gesellix/wait-for postgres:5432

8. Run the integration test, connecting to PostgreSQL by using the following command:

   docker run -e DATABASE="postgresql://postgres:postgres@postgres:5432/pos sslmode=disable" --link postgres onuryilmaz/book-

server/integration-test:latest

You will obtain the following output:

```
 /cloud-native $ docker run -e DATABASE="postgresql://postgres:postgres@postgres:5432/postgres
?sslmode=disable" --link postgres onuryilmaz/book-server/integration-test:latest
?       gitlab.com/onuryilmaz/book-server/cmd    [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/books      [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/commons    [no test files]
=== RUN   TestBookServerIntegration
time="2018-11-19T15:38:08Z" level=info msg="Starting REST server..."
time="2018-11-19T15:38:08Z" level=info msg="REST server connecting to port 41399"

  Start and check RESTServer ✓✓
    Ping server ✓✓✓✓
    Init books database ✓✓✓✓
    Get books ✓✓


12 total assertions

--- PASS: TestBookServerIntegration (3.10s)
PASS
ok      gitlab.com/onuryilmaz/book-server/pkg/server    3.111s
 /cloud-native $
```

Figure 2.8: Output of the integration test for PostgreSQL database

As expected, book-server passed the integration test against the PostgreSQL database.

9. Stop and remove the PostgreSQL database by running the following command:

   docker stop postgres

   docker rm postgres

10. Start and wait for the MSSQL database by running the following command:

    docker run -d -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=Pass_word" -p 1433:1433 --name mssql mcr.microsoft.com/mssql/server:2017-latest

    docker run --rm --link mssql:mssql gesellix/wait-for mssql:1433

11. Run the integration test, connecting to MSSQL by using

the following command:

docker run -e DATABASE="mssql://sa:Pass_word@mssql:1433" --link mssql onuryilmaz/book-server/integration-test:latest

You will obtain the following output:

```
 /cloud-native $ docker run -e DATABASE="mssql://sa:Pass_word@mssql:1433" --link mssql onuryil
maz/book-server/integration-test:latest
?       gitlab.com/onuryilmaz/book-server/cmd    [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/books      [no test files]
?       gitlab.com/onuryilmaz/book-server/pkg/commons    [no test files]
=== RUN   TestBookServerIntegration
time="2018-11-19T15:38:27Z" level=info msg="Starting REST server..."
time="2018-11-19T15:38:27Z" level=info msg="REST server connecting to port 34313"

  Start and check RESTServer ✔✔
    Ping server ✔✔✔✔
    Init books database ✔✔✔✔
    Get books ✔✔


12 total assertions

--- PASS: TestBookServerIntegration (3.10s)
PASS
ok      gitlab.com/onuryilmaz/book-server/pkg/server    3.107s
 /cloud-native $
```

Figure 2.9: Output of the integration test for MSSQL database

As expected, **`book-server`** also passed the integration test against the MSSQL database.

12. Stop and remove the MSSQL database using the following:

    docker stop mssql

    docker rm mssql

    All three results of the integration tests show that 12 assertions were completed successfully for each run. This indicates that the REST API server is working against all supported SQL databases without any obstacles.

Until now, we have described static code analysis for finding

the issues in the source code. Then, unit testing was performed to test functionalities in isolation. Following that, smoke testing was conducted before integration testing. Finally, comprehensive integration testing was performed against all supported environments for all features. These tests ensure that the applications are working as expected and designed. In the next section, we will focus on building the cloud-native applications as executables.

Building Cloud-Native Applications

CI does not only test applications but also creates deliverables such as executable binaries or UI packages. With the cloud-native applications, CI pipelines are expected to create production-ready containers as deliverables. Containers should already include the required libraries and binaries, and they should be ready to consume external volumes, environment variables, or configurations during runtime. With Docker runtime and tooling, it is easy to build and deliver containers; however, there are three critical points for the containers of cloud-native applications:

Single Application Packaging: Container runtime and the orchestrators such as Kubernetes work best when only one application is packaged to run in one container. For instance, only **`book-server`** should run in the container, and it should gracefully close its network resources when **`SIGTERM`** signals are received, as shown in the following code. Signal handling and the graceful shutdown of resources can be checked in **`cmd/main.go`** in **`book-server`**:

```
func main() {
```

```
//[...]

sigs := make(chan os.Signal, 1)

//[...]

signal.Notify(sigs, os.Interrupt, syscall.SIGTERM)

//[...]

webserver.Start()

<-sigs

//[...]

webserver.Stop()

}
```

In the partial main function provided in the preceding code, Go channels are utilized for asynchronous waiting. Since it is unknown when we will receive the **`SIGTERM`** signal to close the applications, the sigs channel is created and registered for **`SIGTERM`**. It is critical to prepare yourself for handling the signals before actually starting the web server. This is because the application needs to close the server gracefully when these signals are received. With the retrieval of **`SIGTERM`**, the registered sigs channel will be invoked, and the code will continue with the lines starting **`<-sigs`** to gracefully close the web server.

When both the database server and the REST API server are running in the same container, there will be multiple **`parent`**

`processes` running in the same container. This makes it difficult to manage the container life cycle and debug if there are problems.

Optimized Docker Builds: Building the containers is one of the most time-consuming stages of CI pipelines when they are not optimized. Docker build operations are based on caching layers of the images; therefore, the steps in the Dockerfile should be optimized to use caches efficiently. The most common example of this is to avoid installing dependencies at each run, as follows:

```
FROM node

WORKDIR app

COPY . .

RUN npm install

CMD [ "npm", "start" ]
```

This container definition shows the natural flow of a Node.js development: first, copying all of the resources to the `app` folder, installing dependencies, and finally running the application. Docker works with a layered filesystem and checks whether it can reuse the cache between `Dockerfile` steps. For instance, if the source code has not changed, it will not be copied in line three of the preceding code, and the remaining commands will not be executed. However, whenever the source code changes, line three will be executed as well as `npm` install, which installs all dependencies. When the practical development environment is considered, there is

no need to install dependencies with every source code change. Therefore, dependency handling and source code are separated in the following **`Dockerfile`**:

```
FROM node

WORKDIR app

COPY package.json .

RUN npm install

COPY . .

CMD [ "npm", "start" ]
```

In the updated Dockerfile, **`package.json`**, where the dependencies are defined, is copied earlier, followed by the **`npm install`** stage. Docker checks whether the files have changed during every build "in order" to utilize cache layers. If the requirements in the **`package.json`** haven't changed, Docker will not install the dependencies in each run and just continue with copying the source code. These optimizations in the usage of Docker cache layers helps to decrease pipeline durations without losing any functionality.

Smallest Container Images: Cloud-native applications are expected to run at a large scale, so the containers should be lightweight to deploy, move, and manage. Therefore, it is crucial to have the smallest container images to run in production. There are two critical points for minimizing container images:

- Don't include any additional tools such as debuggers,

compilers, or `vim` that you may feel necessary for debugging or configuration.

- Use the multi-stage build capability of Docker to isolate binaries from the development environment. For instance, to create a Go binary, Go tooling should be installed in the container. However, when the binary is generated, there is no need for Go tooling in production.

In the next exercise, the effect of the development environment regarding container image size will be demonstrated.

EXERCISE 6: CREATING MULTI-STAGE DOCKER BUILDS

In this exercise, the `book-server` microservice will be built and the resulting container images will be checked. To complete this exercise, the following steps have to be executed:

1. Open the root folder of the book-server API and check the multistage Dockerfile:

   FROM golang:alpine as builder

   ADD . /go/src/gitlab.com/onuryilmaz/book-server

   WORKDIR /go/src/gitlab.com/onuryilmaz/book-server/cmd

   RUN go build -o book-server

   FROM alpine as production

```
COPY --from=builder
/go/src/gitlab.com/onuryilmaz/book-server/cmd/book-
server /book-server

ENTRYPOINT ["/book-server"]
```

The preceding code is an example of a multi-stage Docker build file with the **`builder`** and **`production`** stages. In the builder stage, the Go development environment is used and an executable binary is generated. In the **`production`** stage, the executable is copied from the **`builder`** stage.

2. Build the images for two different targets of builder and production by running the following code:

```
docker build --target builder -t onuryilmaz/book-
server/build:latest .

docker build --target production -t onuryilmaz/book-
server/production:latest .
```

With the successful run of these commands, two Docker images will be created by two different targets of **`builder`** and **`production`**, as defined in the Dockerfile in step 1.

3. Check the size of the images with the docker images command:

```
docker images onuryilmaz/book-server/*
```

You will obtain the following output:

Figure 2.10: Output for the size of the images

In the **`build`** image, the development, build environment, and additionally the **`book-server`** executable will make a total of **`335 MB`**. On the other hand, when we use a leaner operating system and copy only the **`book-server`** executable, the total size of the image becomes **`14 MB`**. Since these images will be pushed and pulled multiple times by CI pipelines and container orchestration schedulers, it is crucial to minimize container image sizes. With slimmer container images, the download and start time of microservices decreases substantially, leading to better scalability and reliability.

In this section, three essential points for building cloud-native applications are discussed as single package applications for optimizing Docker builds and creating smaller images. In the following section, a structured transition plan will be discussed for cloud-native CI design.

CHECKLIST FOR CLOUD-NATIVE CI DESIGN

Adopting and creating a cloud-native CI system is not straightforward and requires a high level of effort. Therefore, it is beneficial to develop a feasible plan by analyzing the current system and setting a target design. In this section, three levels of adoption and transition are constructed, as follows:

Beginner	Intermediate	Advanced
• Source code version control • Test scripts • Build scripts • Scheduled builds and manual versioning • Build environment • Test environment	• Feature and bugfix branches • Automated test running for pull-requests • Automated builds for master branch • CI server with build and test workers	• Automated tag and versioning • Metric collection over automated tests • Optimized container images • Alerts and notifications for test results

Figure 2.11: Plan for different levels of adoption

The Beginner stage aims to create awareness about how to test and build the microservices by storing the code in a versioning system and creating scripts. It is essential to create build and test environments to decide and arrange the required libraries or operating systems. In the Intermediate level, automation should be started with a CI server with the build and test workers. It is crucial to automatically check pull-requests before merging and automatically building new packages after the merge. In the Advanced level, CI pipelines should be well-established to collect data and automatically create new packages. In addition, it is beneficial to generate alerts and notifications from tests. The levels and starting points for each organization could be different; however, a similar transformation plan helps to achieve the cloud-native CI for the applications of the future.

Different levels of testing, starting with static code analysis to check for potential problems in the source code to integration testing, which includes running multiple services, was presented. With the successful test results, the best practices for building cloud-native microservices was discussed. In the following activity, all of these steps will be converted into a CI

pipeline for the `book-server` application.

ACTIVITY 1: BUILDING A CI PIPELINE FOR CLOUD-NATIVE MICROSERVICES

You have been tasked with creating a CI pipeline for a `book-server` application that is designed to be built and tested with containers.

To complete this activity, all previous exercises across all chapters will need to be completed. Use the test and build functions from the previous exercises with the help of GitLab CI/CD pipelines. Once completed, you should have a complete pipeline, starting with static code analysis, passing all tests, and finally building the production-ready container. The pipeline stages and their statuses should be checked from the GitLab web interface:

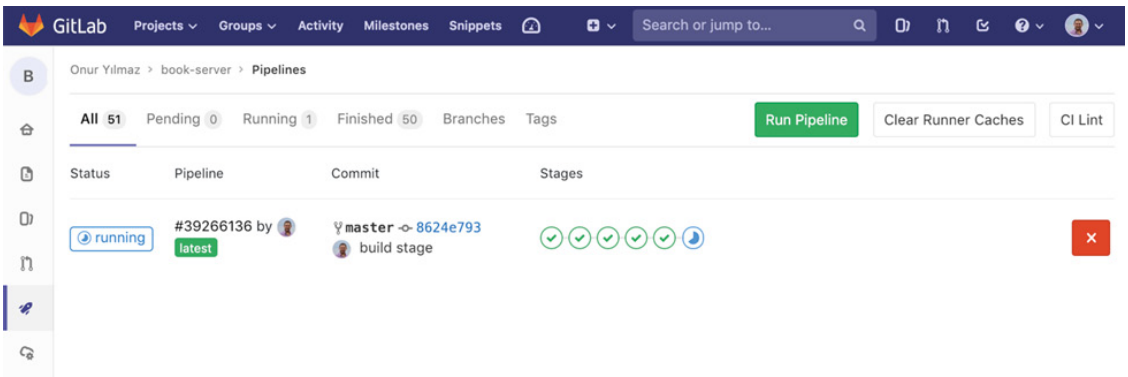

Figure 2.12: CI/CD Pipelines view on GitLab

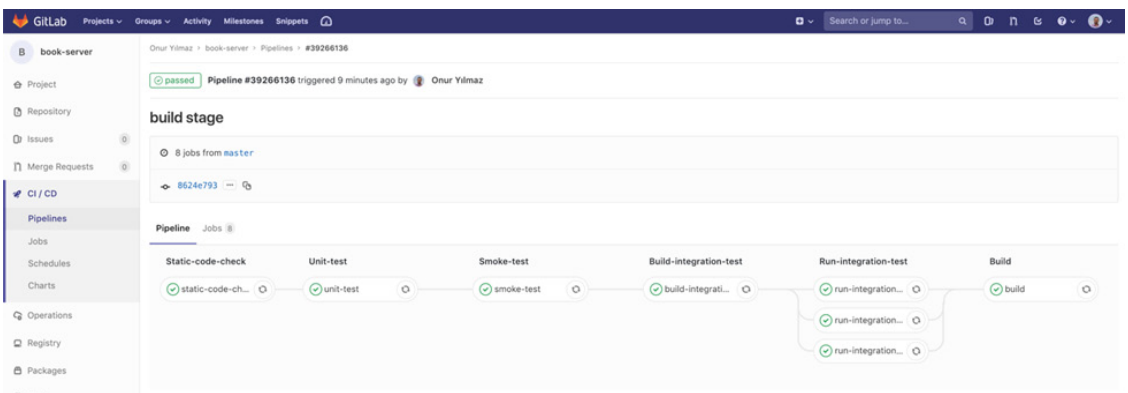

Figure 2.13: Pipeline stages on GitLab

Perform the following steps to complete this activity:

1. Fork the book-server code to your GitLab project from the **`book-server`** repository (https://gitlab.com/TrainingByPackt/book-server).
2. Delete the existing code in the **`.gitlab-ci.yml`** file. We will be creating all of the stages in this file in the following steps. You should already have the test and build Dockerfiles, along with the source code in the repository once you fork it to your own namespace.
3. Define the following stages in the **`.gitlab-ci.yml`** file to run in Docker-in-Docker, namely the **`docker:dind`** service: static-code-check, unit-test, smoke-test, build-integration-test, run-integration-test, and build.
4. Create the **`static-code`** check stage.
5. Create the **`unit-test`** stage.
6. Create the **`smoke-test`** stage.
7. Create the **`build-integration-test`** check stage.
8. Create the **`run-integration-test`** check stage.
9. Create the **`build`** stage.
10. Commit the **`.gitlab-ci.yml`** file to the repository.
11. Click the **CI/CD** tab on the GitLab interface and then click on the **Run Pipeline** tab.
12. Click on **Create pipeline** to build the CI/CD pipeline.

Note

The solution to this activity can be found on page 118

Summary

In this chapter, we discussed CI as the first DevOps practice for cloud-native applications. First, container technology was presented, since it is the building block of cloud-native applications. Its history and technological background from a Linux perspective was explained so that the students can design applications by efficiently utilizing container capabilities. Thereafter, we conducted the testing of cloud-native applications, beginning with source code analysis to find flaws in the code. Each functionality was tested in isolation via unit tests.

Smoke tests were used as a checkpoint to decide whether the microservices were still working when they were connected to other microservices. Finally, integration tests were run to cover all functionalities against all supported development environments. All of the testing stages of cloud-native applications were designed and run inside containers, with isolation and scalability in mind. With successful test results, we then built cloud-native applications as lightweight containers. To create slim containers that worked well with cloud providers and orchestrators, the best practices were discussed for building, packaging, and designing applications. We also presented a structured guideline so that we can implement cloud-native CI design with critical checkpoints and achievements. Although each organization will have unique visions and dynamics, it is beneficial to have a general guideline so that the transition from conventional approaches is designed and planned.

In the next chapter, we will continue with continuous delivery and deployment. We will focus on the best practices for delivering and deploying the cloud-native applications that were tested and built in this chapter.

Cloud-Native Continuous Delivery and Deployment

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the fundamentals of continuous delivery for cloud-native architecture
- Identify different versioning schemes for microservices
- Run and use a hosted and secure cloud container registry
- Create a Kubernetes cluster as a cloud-native deployment platform
- Package cloud-native applications using Helm
- Design a continuous delivery and deployment pipeline for a cloud-native microservice

This chapter presents the fundamentals of continuous delivery for cloud-native architecture and several exercises on versioning schemes, Kubernetes, Helm, and so on.

Introduction

Continuous delivery/deployment (CD) is based on the

idea of preparing releases and installing them for the end users as soon as possible in a consistent way. These are the extended practices that are followed after the successful implementation of continuous integration. Although the scope of these practices is different, the overall goal is to deliver robust, reliable, and scalable applications as soon as possible.

Continuous delivery and the related automation steps ensure that the main branch of the repository is ready to be shipped. With the same idea, continuous deployment automatically updates the production environment with the output of CD. In cloud-native architecture, continuous integration ensures that the microservices work as expected, and that the containers are built successfully. Cloud-native CD focuses on versioning and releasing containers to public or private container registries. Finally, continuous deployment handles the installation and updating of the microservices running in cloud platforms such as AWS, Google Cloud, or private systems.

Cloud-native CD pipelines create two critical values for achieving better quality scalable applications:

Incremental releases: Microservices are small-focused services, and simplistic incremental releases of the microservices result in better-managed production environments. As microservices are developed and designed separately, their life cycle for installation and updates should also be managed independently with minimal iterations.

To understand the preceding concept better, let's review an example. Consider an e-commerce application with the frontend, billing, and recommendation engine microservices. Changes in the frontend could be delivered and deployed independently of the billing and recommendation engine.

Since the billing and recommendation engine communicate over fixed and versioned APIs to the frontend, incremental releases of the frontend will not create a disturbance in the production environment. Instead of substantial installation efforts with possible downtimes, building a pipeline that installs additional changes automatically will produce more reliable production environments. In addition, this shortens the feedback loop by minimizing the time between features becoming ready, and features are installed for customer usage.

Minimal Human Interaction/Error: Continuous integration ensures that the microservices are tested and built successfully. If no automation is implemented for the rest of the software's life cycle, then someone should copy the artifacts from the integration, somehow label them with versions, and send them to the customers. Besides, human interaction is needed to install or update the artifacts for the customer environment. In this non-autonomous world with a rich level of human interaction, it is inevitable that you will confront human errors. For instance, it is widespread to see old versions installed to production or corrupt configuration files attached to the containers. With a rigid CD pipeline, human interaction, decision-making, and efforts are minimized for installing and updating applications.

For the goal of delivering and deploying cloud-native applications as early as possible, implementing continuous delivery and deployment practices in a cloud-native way is critical. In this chapter, first, the delivery of containers is discussed by versioning and releasing container images. Following that, cloud-native deployment strategies are discussed so that we can install and update microservices automatically. Finally, a checklist for the cloud-native CD

pipeline design is presented.

Note

In this chapter, the **`book-server`** *from the previous chapter will be used throughout the examples. However, to efficiently manage CI/CD pipelines between chapters, delivery and deployment-related changes are kept in a separate repository: https://gitlab.com/TrainingByPackt/book-server-cd. The code files can also be found here: https://bit.ly/2QBwOCy.*

Continuous Delivery of Containers

Cloud-native continuous delivery focuses on releasing microservices to customers, packaged as lightweight containers. In this chapter, first, the best practices for versioning containers are explained. Following that, the options to deliver containers are discussed. These two main stages ensure that the output of the continuous integration is labeled and versioned in an appropriate way and made available for customers.

VERSIONING CONTAINER IMAGES

The lightweight containers of microservices are designed to be portable so that they can run on any server in the data center in a fast way. Container orchestrator systems such as Kubernetes or Docker Swarm require a container image name and tag to start microservices. As expected, the name is static and does not change, whereas a tag is used for versioning of the container images. For instance, we can run an **`ubuntu`** instance by using **`ubuntu:16.04`** or **`ubuntu:18.04`** based on our requirements for the **`ubuntu`** version. There are two

mainstream approaches to versioning containers of microservices: semantic versioning and unique tagging.

Semantic Versioning

Semantic versioning is the most common way of versioning applications in software development. It is based on versioning applications with three numbers, formatted as `major.minor.patch`. These three numbers are increased with a new version according to the following rules:

- Increase the major version when incompatible API changes are released. For instance, if the response data for a REST API request in version `1.x.x` changed, the new package should be versioned as `2.0.0`.
- Increase the minor version when new features are proposed, but they do not break the old functionality. If you add a new REST API endpoint for the old ones in version `1.1.x`, clients are still able to use the previous ones, so the application is backward-compatible. However, if any client wants to use the newly developed endpoints, they should use Version `1.2.x`.
- Increase the patch version when only bugfixes are proposed without any new functionality or breaking API changes. For instance, if you fix a bug in version `1.2.3`, you are expected to deliver a new version under the tag of `1.2.4`.

Semantic versioning is widespread and the de facto way of managing versions of applications. However, there are some drawbacks to this approach. The first problem is that semantic versioning mostly requires human interaction to decide when to create a new patch or minor version. The

second problem is that in the long run, there will be less informative gain between the two versions. In other words, it is possible to know what the main difference between versions `10.1.15` and `10.1.16` are if they were shipped last week. However, after a couple of months, it is difficult to remember what has been fixed between these two versions.

Although semantic versioning could be a short-lived, human-intensive effort considering the development and production environment, it is valuable for cloud-native applications when it comes to versioning base images that are not changing for a long time.

Unique Tagging

Unique tagging is based on the idea of attaching a unique version tag to every build artifact. The most common values for uniquely tagging the artifacts are a timestamp, build-ID, `git` commit, and their combinations:

Timestamp: Artifacts are versioned by the timestamp so that we know when they were created. It is common to use a build server time zone or UTC, such as `book-server:20181224-120015` so that we can tell whether the container was built on December 24, 2018 at 12:00:15. It is also possible to use UNIX epoch time, which shows the number of seconds elapsed since January 1, 1970 00:00:00 in UTC. Both timestamp tags are unique, where the UNIX epoch is more machine-friendly than the actual date and time.

Build-ID: Build-IDs are provided by the build servers incrementally so that every artifact will have unique tags. For instance, container images will follow up this scheme when they are tagged with build numbers such as `book-server:101`, `book-server:102`, and so on. This approach

ensures that tags are incremental and unique, although they do not provide information about their build time or content.

Git Commit ID: For every commit made into the source code repository, Git calculates a unique hash number and maintains it for future reference as a commit ID. It is very common to see Git hashes in GitHub or GitLab web interfaces, as well as the command line. For instance, in the output of `git log`, the first line shows the commit hash, which is a 40-character string:

```
 /cloud-native $ git log
commit 977837d04f62c168ab4b3ec9c87dafc4a64410a7 (HEAD -> master, origin/master)
Author: Onur Yilmaz <onur***@***.com>
Date:   Sun Dec 2 19:53:23 2018 +0100

    scale to 10

commit 24a838c4a756faf2445d3746fdbd97e982f5f858
Author: Onur Yilmaz <onur***@***.com>
Date:   Sun Dec 2 19:47:30 2018 +0100

    First commit
 /cloud-native $
```

Figure 3.1: An example of the most recent commits in the book-server repository

In Git, commit hashes are unique, since they are calculated with a cryptographic hash function known as SHA-1 for the contents of the commit. It is a very core functionality of Git, which ensures that no corruption or change could be made within the same commit ID. Commit IDs are also valuable, since they directly show the state of the source code repository. For instance, if `book-server:977..0a7` is running on the cloud, there is no need for additional knowledge to trace back to the source code and find the state of the repository where the image was built. It is valuable in a fast development environment with multiple deployments and updates being made in production.

Note

The Secure Hash Algorithm 1 (SHA-1), which was developed by the US National Security Agency (NSA), is primarily used to verify the content of files by creating a checksum and sending it with the data. When the data reaches the target, the checksum is calculated again using SHA-1 and compared to the original value to check whether the data was changed or corrupted during transfer.

Unique tagging also ensures that the tags are not reused; in other words, there should not be two different container images with the version of `book-server:453f164fc`. This is a critical issue for distributed applications that are scheduled by container orchestrators such as Docker Swarm or Kubernetes. Distributed applications run on many worker hosts where container images are pulled when they are the first to run, and images are reused within the same host.

Let's understand this point with the help of an example. Consider that `book-server:unique` is our uniquely tagged container image, and that our application is running two instances on **Node A** and **Node B**. The `book-server:unique` container image is pulled from a central repository to **Node A** and **Node B**, and our application is running without any problem, as shown in the following diagram:

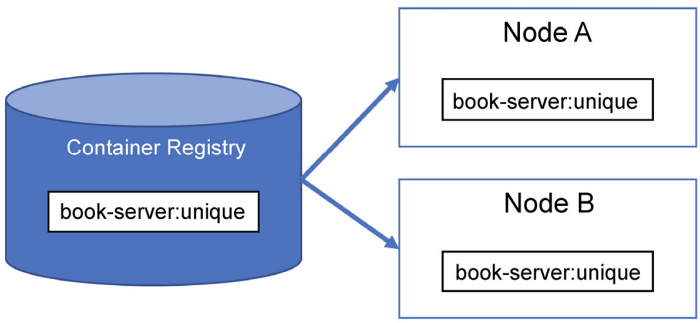

Figure 3.2: Both nodes are running the image that was pulled from the same repository

Unfortunately, we have identified a bug in the code, fixed it, and now want to publish it under the same **`book-server:unique`** tag and upload it to the central repository. Now there are different images at the registry and nodes, as shown in the following diagram:

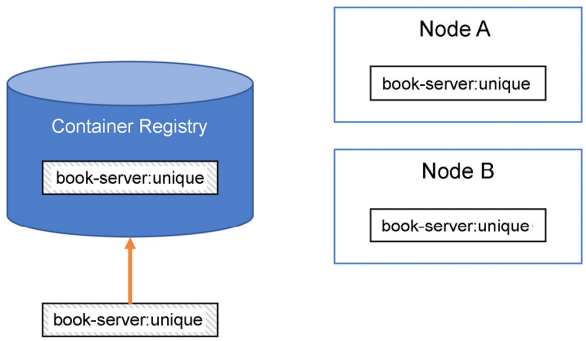

Figure 3.3: The new image is published in the same tag

At the same time, in the cluster, it is expected that some worker nodes go down and schedule their workload to other nodes. Let's assume that **Node B** is faulty, new worker **Node C** joins to the cluster, and our application will now run in Node C. Since there is no **`book-server:unique`** image in **Node C**, it will retrieve it from the repository and run it. Now we have two instances of our application running in the cluster with the **`book-server:unique`** tag, but actually with different content, as illustrated in the following diagram:

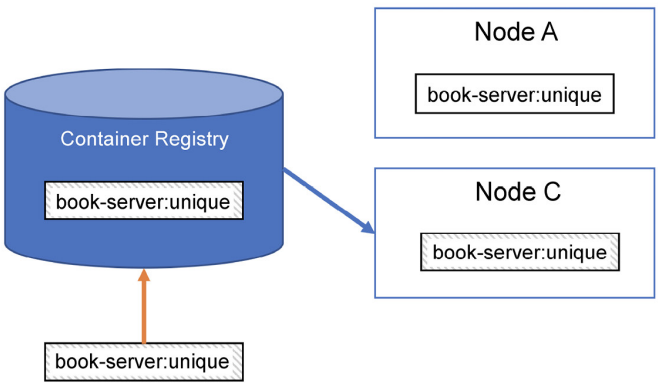

Figure 3.4: The nodes are running different images with the same tag

Now it is possible for some users to detect the bug we have already fixed, and this will create more transparency issues on

what is delivered and what is fixed. In such an inconsistent setup, it is complicated to debug problems and correctly find root causes. Therefore, it is crucial that you create consistent images and not overwrite them if unique tags are implemented.

Considering the gains and drawbacks of different tagging approaches, in the following exercise, we will show you how semantic versions are used for base images and how unique tags are used for the production-ready container images.

EXERCISE 7: VERSIONING DOCKER IMAGES

In this exercise, we will create Docker images that use semantic versioning for base images and unique tags for production-ready images:

1. Open the **`Dockerfile`** in the root folder of **`book-server-cd`** and review the container definitions for building the **`book-server`**:

   FROM golang:1.11.2-alpine3.8 as builder

   ADD . /go/src/gitlab.com/onuryilmaz/book-server-cd

   WORKDIR /go/src/gitlab.com/onuryilmaz/book-server-cd/cmd

   ARG VERSION

   RUN go build -ldflags "-X main.version=$VERSION" -o book-server

   FROM alpine:3.8 as production

   COPY --from=builder

/go/src/gitlab.com/onuryilmaz/book-server-cd/cmd/book-server /book-server

ENTRYPOINT ["/book-server"]

In this file, there are two base images: **`golang:1.11.2-alpine3.8`** for **`builder`** and **`alpine:3.8`** for production. The semantic version of the **`golang`** image shows us that we are using Go 1.11.2 and that it is running on **`alpine`** version 3.8. With the same approach, it is clear that the production image is alpine with Version 3.8. The **`VERSION`** parameter is defined as ARG, and it is used in the go build line. It allows us to pass the version parameter so that the resulting executable also knows its version.

2. Obtain the unique **`git`** commit ID by running the following code in the shell:

   export VERSION=$(git rev-parse --verify HEAD)

3. Review the obtained version by printing out the terminal using the following command:

   echo $VERSION

   $ 977837d04f62c168ab4b3ec9c87dafc4a64410a7

4. Build the image with the version from the last step using the following command:

   docker build --target production --build-arg VERSION=$VERSION -t book-server/production:$VERSION .

   You will see the following output:

```
/cloud-native $ docker build --target production --build-arg VERSION=$VERSION -t book-server/production:$VERSION .
Sending build context to Docker daemon  11.98MB
Step 1/8 : FROM golang:1.11.2-alpine3.8 as builder
 ---> 57915f96905a
Step 2/8 : ADD . /go/src/gitlab.com/onuryilmaz/book-server-cd
 ---> 45e3ea645895
Step 3/8 : WORKDIR /go/src/gitlab.com/onuryilmaz/book-server-cd/cmd
 ---> Running in 3d44307beeb5
Removing intermediate container 3d44307beeb5
 ---> a3826e526355
Step 4/8 : ARG VERSION
 ---> Running in 6c7b865a01d9
Removing intermediate container 6c7b865a01d9
 ---> 1cf86d4e11b4
Step 5/8 : RUN go build -ldflags "-X main.version=$VERSION" -o book-server
 ---> Running in 79db76ee7600
Removing intermediate container 79db76ee7600
 ---> 3578c6f50151
Step 6/8 : FROM alpine:3.8 as production
 ---> 196d12cf6ab1
Step 7/8 : COPY --from=builder /go/src/gitlab.com/onuryilmaz/book-server-cd/cmd/book-server /book-server
 ---> 00029b3e317f
Step 8/8 : ENTRYPOINT ["/book-server"]
 ---> Running in 5f276b9ab54b
Removing intermediate container 5f276b9ab54b
 ---> 57dddfed2566
Successfully built 57dddfed2566
Successfully tagged book-server/production:977837d04f62c168ab4b3ec9c87dafc4a64410a7
 /cloud-native $
```

Figure 3.5: Container build for production

As you can see from the preceding output, we have successfully built and tagged our **`book-server`** with the version that we obtained in step 2.

5. Check the images and tags by running the following code:

   docker images book-server/production

   You will obtain the following output:

```
 /cloud-native $ docker images book-server/production
REPOSITORY                  TAG                                        IMAGE ID            CREATED              SIZE
book-server/production      977837d04f62c168ab4b3ec9c87dafc4a64410a7   57dddfed2566        About a minute ago   14.1MB
 /cloud-native $
```

Figure 3.6: List of container images

The preceding output lists the **`book-server/production`** images and shows us that there is a tag with the unique **`git`** commit ID that we retrieved in step 2.

Semantic versioning of the base images makes it easier to follow up changes in the infrastructure that we are using. For instance, by checking the **`Dockerfile`**, we can see the Go version for the build and can manually iterate when changes are required. It is not expected that you automatically increase development environment versions, like in the base image, since they could also need changes in source code and

implementation.

Production images should be machine-friendly for automation inside CI/CD pipelines. Therefore, we have used unique tagging for production-ready images with the help of Git commit hashes. In the following section, we will discuss how to deliver these production-ready images to the customers.

DELIVERING CONTAINER IMAGES

The containers of the microservices have been tested, built, and versioned, and now it is time to deliver these images in a cloud-native way. Although there are ways to provide container images in conventional means, we will focus on delivering in a cloud-native and secure method. For instance, it is possible to package a Docker container image as **`tar`** by using the **`docker save`** command and unpackaging it on the customer side's by using **`docker load`**; however, it is inefficient to automate and scale.

When the continuous build and delivery of containers are thoughtful and natural, the most common solution is to use a Docker registry. Docker registry is a content delivery and storage container for Docker images. Essentially, a Docker image consists of layers, where each layer indicates a filesystem difference. Images are tagged and given specific versions, and different tags of the same Docker image are kept in the same repository. These elements are used in all Docker commands, as follows: **`registry/repository:tag`**.

Docker registries play a crucial role in continuous delivery and deployment. They make it possible to run hundreds of instances in a distributed cluster by storing them efficiently and delivering them in a scalable fashion. Without a registry,

whenever a new container image is built, it should be distributed over `SSH`, `SCP`, or any other protocol to all worker nodes. Instead of this push model, the container registry delivers the containers via the on-demand approach. It is possible to run a self-hosted local registry and maintain it in-house, as well as using the cloud registries that are publicly available.

The self-hosted registry provides more flexibility in where to store and distribute images; however, when it is open to customers or third-party access, its security is critical. Cloud registries provide high-level security features that could work for startups and large enterprises. There are various cloud registry services, and some of the most popular ones are as follows:

- Docker Hub: https://hub.docker.com/
- Quay: https://quay.io/
- AWS EC2 Container Registry: https://aws.amazon.com/ecr/
- Google Container Registry: https://cloud.google.com/container-registry/

In the following exercise, we will start by running a local registry and demonstrate how Docker works with our new registry. Then, in Exercise 3, we will work against a more secure cloud Docker registry running in GitLab.

EXERCISE 8: USING A SELF-HOSTED DOCKER REGISTRY

1. Create a Docker registry locally by running the following command in a terminal:

docker run -p 5000:5000 --name registry registry:2

The following output is obtained:

```
 /cloud-native $ docker run -p 5000:5000 --name registry registry:2
time="2018-12-02T20:34:34Z" level=warning msg="No HTTP secret provided - generated random secret. T
his may cause problems with uploads if multiple registries are behind a load-balancer. To provide a
 shared secret, fill in http.secret in the configuration file or set the REGISTRY_HTTP_SECRET envir
onment variable." go.version=go1.7.6 instance.id=6fab7e9a-dfc6-44c7-b8b5-e954a3d2da15 version=v2.6.
2
time="2018-12-02T20:34:34Z" level=info msg="redis not configured" go.version=go1.7.6 instance.id=6f
ab7e9a-dfc6-44c7-b8b5-e954a3d2da15 version=v2.6.2
time="2018-12-02T20:34:34Z" level=info msg="Starting upload purge in 51m0s" go.version=go1.7.6 inst
ance.id=6fab7e9a-dfc6-44c7-b8b5-e954a3d2da15 version=v2.6.2
time="2018-12-02T20:34:34Z" level=info msg="using inmemory blob descriptor cache" go.version=go1.7.
6 instance.id=6fab7e9a-dfc6-44c7-b8b5-e954a3d2da15 version=v2.6.2
time="2018-12-02T20:34:34Z" level=info msg="listening on [::]:5000" go.version=go1.7.6 instance.id=
6fab7e9a-dfc6-44c7-b8b5-e954a3d2da15 version=v2.6.2
```

Figure 3.7: Docker registry start logs

By using the preceding command, we are running a Docker container from the **`registry:2`** image. The name of the container is **`registry`**, and it publishes the **`5000`** port to our localhost. The last line of the output shows that it started to listen on the **`5000`** port for incoming requests.

Note

Ensure that the preceding terminal is kept open. As we progress with this exercise, we will be observing the output on this terminal.

2. Open another terminal and check for the **`book-server`** images from the last section by running the following command:

   docker images book-server/production

   You will see the following output:

```
 /cloud-native $ docker images book-server/production
REPOSITORY                  TAG                                        IMAGE ID            CREATED              SIZE
book-server/production      977837d04f62c168ab4b3ec9c87dafc4a64410a7   57dddfed2566        About a minute ago   14.1MB
 /cloud-native $
```

Figure 3.8: List of container images

It is expected that you see the image that was built in the first exercise with its unique tag; this image will be used in the following steps.

3. Tag the existing image with the new registry address and push it to the new registry using the following code:

   docker tag book-server/production:977837d04f62c168ab4b3ec9c87dafc4a64410a7 localhost:5000/book-server:latest

   docker push localhost:5000/book-server:latest

   You will obtain the following output:

```
 /cloud-native $ docker tag book-server/production:977837d04f62c168ab4b3ec9c87dafc4a64410a7 \
> localhost:5000/book-server:latest
 /cloud-native $ docker push localhost:5000/book-server:latest
The push refers to repository [localhost:5000/book-server]
10ed9de275d6: Pushed
df64d3292fd6: Pushed
latest: digest: sha256:30bf5d7fd6b8c7e2e2b4cd89f1861956bf2119abeef4eb5e07ec20f405fe3860 size: 739
 /cloud-native $
```

Figure 3.9: Uploading the container to the registry

   The preceding output shows that the image was uploaded to the registry from our local system.

4. Clear the tagged image from Docker using the following command:

   docker rmi localhost:5000/book-server:latest

   This command will not create output, and it will remove the Docker image with the provided tag. We do this so that we can test that our registry can distribute the image.

5. Run an instance of **`book-server`** using the following code:

   docker run localhost:5000/book-server:latest

The following output can be seen on the command prompt:

```
 /cloud-native $ docker run  localhost:5000/book-server:latest
Unable to find image 'localhost:5000/book-server:latest' locally
latest: Pulling from book-server
Digest: sha256:30bf5d7fd6b8c7e2e2b4cd89f1861956bf2119abeef4eb5e07ec20f405fe3860
Status: Downloaded newer image for localhost:5000/book-server:latest
time="2018-12-02T20:37:36Z" level=warning msg="Database address is empty, some functionality may no
t work.."
time="2018-12-02T20:37:36Z" level=info msg="Starting REST server..."
time="2018-12-02T20:37:36Z" level=info msg="REST server connecting to port 8080"
```

Figure 3.10: Start logs of the book-server

Logs starts by downloading the image from our new registry and then actually starting the **`book-server`**. It is important to mention that this Docker registry is open to access and that it lacks security features.

6. Stop the **`book-server`** and then the registry containers running in the terminal by typing **`ctrl+c`** on the command line.

This exercise shows us how it is easy to start a Docker registry using only Docker containers and that all tooling integrates and works out-of-the-box. However, there is no security and protection over the containers in the registry. In the following exercise, we will work with a more secure cloud Docker registry in GitLab.

EXERCISE 9: USING A SECURE CLOUD DOCKER REGISTRY

1. Fork the **`book-server`** code to your GitLab project from the **`book-server-cd`** (https://gitlab.com/TrainingByPackt/book-server-cd) repository by clicking the Fork button, as shown in the following screenshot:

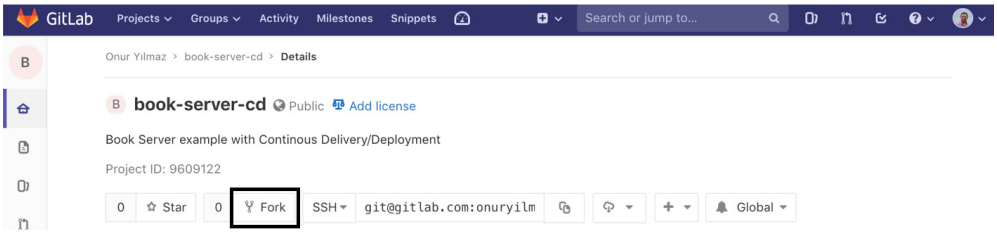

Figure 3.11: Forking the repository on GitLab

2. Open the GitLab interface and go to user settings. Click your user avatar in the top-right corner, choose **Settings**, and then choose **Access tokens**, as shown in the following screenshot:

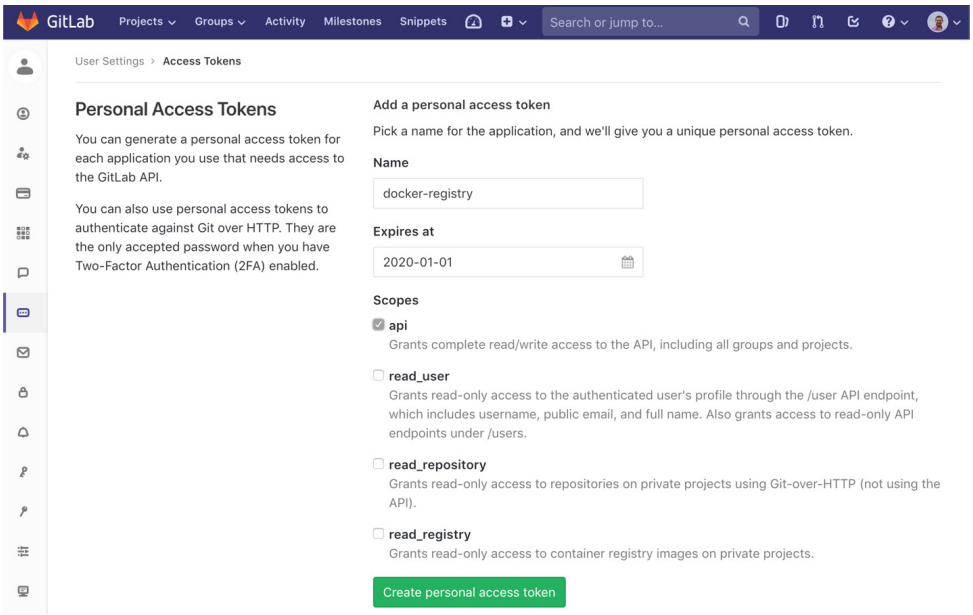

Figure 3.12: Creating access tokens on GitLab

We are performing these steps as we are aiming to securely connect to and interact with the registry, for which an access token is required.

3. Fill the **`name`** and optional expiry date and ensure that **`api`** is selected from the scopes. The **`api`** scope enables both pulling and pushing to the container registry. Click **Create personal access token** and copy the token that is created:

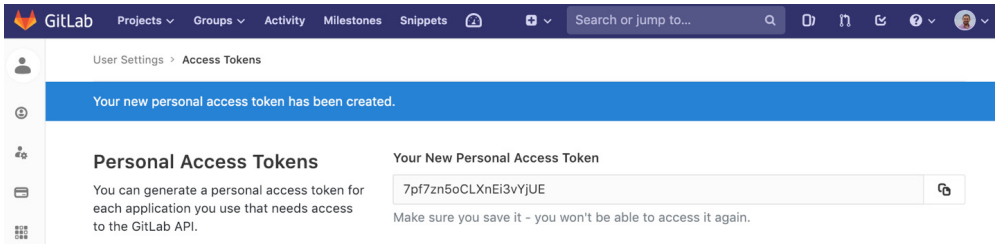

Figure 3.13: Access token created

4. Log into the GitLab registry from the terminal using the following command:

   docker login -u <USERNAME> -p <PERSONAL_ACCESS_TOKEN> registry.gitlab.com

   The following output will be obtained:

```
 /cloud-native $ docker login -u onuryilmaz -p 7Y********jH registry.gitlab.com
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Login Succeeded
 /cloud-native $
```

Figure 3.14: Docker login to GitLab

5. Tag the existing image with the GitLab registry address and push it using the following command:

   docker tag book-server/production:977837d04f62c168ab4b3ec9c87dafc4a64410a7
   docker push registry.gitlab.com/<USERNAME>/<PROJECT_NAMI

   You will obtain the following output:

```
 /cloud-native $ docker tag book-server/production:977837d04f62c168ab4b3ec9c87dafc4a64410a7 \
> registry.gitlab.com/onuryilmaz/book-server-cd:latest
 /cloud-native $ docker push registry.gitlab.com/onuryilmaz/book-server-cd:latest
The push refers to repository [registry.gitlab.com/onuryilmaz/book-server-cd]
10ed9de275d6: Pushed
df64d3292fd6: Layer already exists
latest: digest: sha256:30bf5d7fd6b8c7e2e2b4cd89f1861956bf2119abeef4eb5e07ec20f405fe3860 size: 739
 /cloud-native $
```

Figure 3.15: Uploading the container to the registry

   The preceding output shows us that the image has been uploaded to the GitLab registry from our local system.

6. Check the image from the GitLab web interface under the

Registry tab inside the project, as shown in the following screenshot:

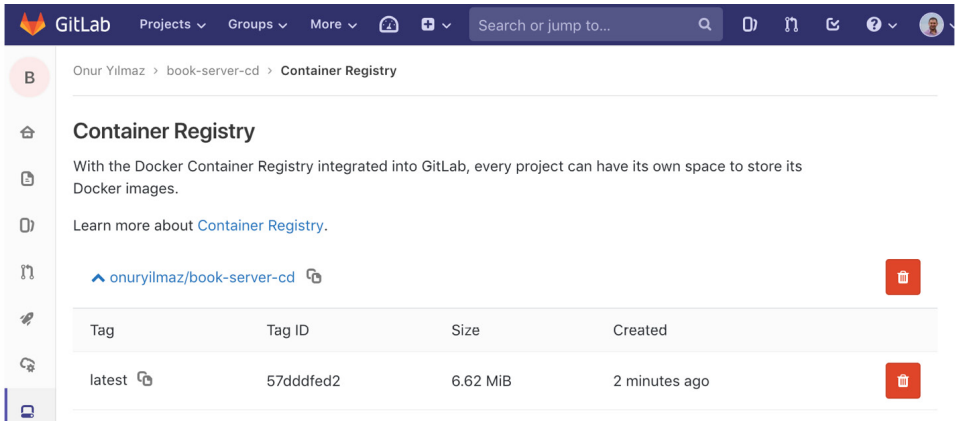

Figure 3.16: Registry and containers on GitLab

As we can see, the GitLab registry has the Docker image with the `latest` tag, which was pushed in step 5.

7. Clear the tagged image from Docker so that we can test whether we can pull the image from the GitLab registry by running the following command on the shell:

   docker rmi registry.gitlab.com/<USERNAME>/<PROJECT_NAME>:latest

8. Run the following command to download the image and begin the application:

   docker run registry.gitlab.com/<USERNAME>/<PROJECT_NAME>:latest

   You will obtain the following output:

```
 /cloud-native $ docker run registry.gitlab.com/onuryilmaz/book-server-cd:latest
Unable to find image 'registry.gitlab.com/onuryilmaz/book-server-cd:latest' locally
latest: Pulling from onuryilmaz/book-server-cd
Digest: sha256:30bf5d7fd6b8c7e2e2b4cd89f1861956bf2119abeef4eb5e07ec20f405fe3860
Status: Downloaded newer image for registry.gitlab.com/onuryilmaz/book-server-cd:latest
time="2018-12-02T20:16:33Z" level=warning msg="Database address is empty, some functionality may no
t work.."
time="2018-12-02T20:16:33Z" level=info msg="Starting REST server..."
time="2018-12-02T20:16:33Z" level=info msg="REST server connecting to port 8080"
```

Figure 3.17: Download and start logs of book-server

As we can see, the logs start by downloading the image from the GitLab registry and then actually starting the book-server as expected.

In this section, we have discussed the best practices for versioning container images and then uploading them to the container registry. By following these guidelines, it is possible to deliver scalable and reliable containerized microservices. In the following section, we will focus on how to deploy and update the cloud-native applications we have just delivered.

CLOUD-NATIVE CONTINUOUS DEPLOYMENT

Cloud-native continuous deployment focuses on configuring, installing, and updating microservices in cloud environments. In this section, first, the cloud environment where the cloud-native applications are deployed will be discussed. Then, how the applications and configurations are packaged for installation is explained. Finally, reliable and scalable cloud-native deployment strategies are presented. These stages ensure that the containerized microservices are configured, installed, and updated in a cloud-native way.

There are many container orchestration tools on the market such as **Mesos**, **Docker Swarm**, **Amazon Elastic Container Service**, and **Kubernetes**. All of these tools have an active community, and many organizations adopt them. However, Kubernetes puts itself forward between others with Google support, a significant amount of popularity, and many success stories, including **GitHub**, **GoDaddy**, and **Workday**. Thus, in this section, we will focus on Kubernetes as a cloud-native CD environment. Besides, the application will be packaged and maintained with the best practices for Kubernetes. In the following section, the characteristics of the Kubernetes environment are discussed,

and a cluster is provisioned that we can use throughout this chapter.

KUBERNETES

Kubernetes is the upcoming and prominent open source container orchestration system that was designed by **Google**. Its fundamental features include the automation, scaling, and scheduling of containerized applications. To have a scalable and reliable cloud-native application, Kubernetes is a crucial part of the toolset. All levels of companies, from start-ups to large enterprises, are using Kubernetes to install and manage cloud-native applications. The essential characteristics of Kubernetes can be listed as follows:

Container orchestration: Kubernetes can schedule and manage containers on multiple worker nodes. In Kubernetes, core services such as **`Kubernetes API server`** and **`Kubernetes schedulers`** are expected to run in master nodes, and all other worker nodes run applications. Kubernetes creates an abstraction for worker nodes so that you do not need to manage how your application is distributed over the cluster. Also, Kubernetes utilizes all the features of container runtimes. For instance, if you can run your application in Docker, you can easily port it to run in Kubernetes without losing any functionality.

Scalability: Scaling applications on the fly based on resource usage or customer demand is possible. In Kubernetes, there are two levels of scalability. First, applications can be scaled with their resource usage. For instance, let's imagine you are running only one **`backend`** instance and you don't want it to use more than **`512 MB`** of memory. Kubernetes will automatically create a second instance of the **`backend`** when

the memory limit is reached. Secondly, Kubernetes cluster can be automatically scaled with new worker nodes when the resource usage in the node is high. Kubernetes could request new worker nodes from the cloud infrastructure provider, and new nodes would join to the cluster to spread the application load. In both directions, scalability is an essential part of Kubernetes, which makes it the platform of scalable and reliable cloud-native applications.

Declarative: All Kubernetes resources are explicitly defined, and Kubernetes ensures that they are running as described. Kubernetes resources are designed to describe the end state. For instance, if there is a need for reverse proxy in the system and it is decided that you should use the popular **`nginx`** with three replicas, the first thing is to write a deployment definition properly for Kubernetes. When **`nginx-deployment`** is checked, it is explicitly mentioned that there will be three replicas of **`nginx:1.15.4`** with the published port of **`80`**:

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

  labels:

    app: nginx

spec:

  replicas: 3
```

```
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
        ports:
        - containerPort: 80
```

When this deployment definition is applied by the **`kubectl apply`** command, Kubernetes will try to have three instances of **`nginx`** with the specified version. This feature allows various clients and applications to work with Kubernetes by declaratively defining and changing the resources.

Storage: Not only stateless applications but also stateful applications that write their state to disk can work in Kubernetes. Containers and microservices are assumed to run only stateless applications; however, storage and volumes are Kubernetes solutions that can run stateful applications. For

instance, you can run all popular databases, such as **MySQL** or **PostgreSQL** in Kubernetes by using their officially supported Helm charts. When they are deployed, Kubernetes will provision volumes from infrastructure providers like **AWS Cloud Storage** or **Google Cloud Storage**. These volumes will be mounted to containers by Kubernetes, and the applications will persist their state while they are running in the cloud.

Updates: Applications are updated smoothly, and cloud-native update strategies are already included in Kubernetes. With the right subset of Kubernetes resources and choosing an appropriate deployment strategy, scalable and reliable cloud-native applications are feasible. For instance, the rolling update upgrades the instances without any downtime. This is the default strategy for updating Kubernetes deployments. In addition, recreate, blue-green, or A/B testing strategies could be implemented in Kubernetes.

Self-healing: Kubernetes tries to achieve the state defined in the declarative resources with the help of self-healing mechanisms. Kubernetes checks containers by their defined health-checks so that they can restart, reschedule, or replace the container automatically. For instance, in the **`book-server`** application, a **`/ping`** endpoint is defined for pinging the web server. With a **`liveness probe`** defined for the HTTP request to **`/ping`**, Kubernetes will continuously check that the web server is up and running and will take action when it does not respond. Self-healing of Kubernetes enables having reliable and scalable applications running in the cloud with minimum human interaction.

Kubernetes can run on various platforms such as Raspberry Pi devices, bare-metal servers, or virtual machines. Each

platform has its benefits and drawbacks considering the price, uptime, and custom needs. The most common Kubernetes solutions could be grouped as follows:

- **Local Solutions**: minikube (https://kubernetes.io/docs/setup/minikube/) and microk8s (https://microk8s.io/)
- **Hosted Solutions**: Amazon Elastic Container Service for Kubernetes (https://aws.amazon.com/eks/)
- **Azure Kubernetes Service** (**AKS**) (https://azure.microsoft.com/en-us/services/kubernetes-service/), Google Kubernetes Engine (https://cloud.google.com/kubernetes-engine/), and Openshift (https://www.openshift.com)
- In the following exercise, we will create a Kubernetes cluster from **Google Kubernetes Engine** (**GKE**). In GKE, master nodes and Kubernetes core services are managed by Google Cloud without any additional price. Only the cost for the worker nodes, where our applications will run, is billed.

EXERCISE 10: CREATING A KUBERNETES CLUSTER

In this exercise, we will create a cluster in GKE and connect it to GitLab for a CI/CD pipeline. Before attempting this exercise, you need to register on the Google Cloud Platform.

Note

If you are using Google Cloud for the first time, you can activate credit to explore Google Cloud products, as mentioned in the header. Google Cloud will need your billing address and payment data so that it can use this after the credit you already have is consumed.

Perform the following steps to complete this exercise:

1. Log into the Google Cloud Console with your Google account: https://console.cloud.google.com:

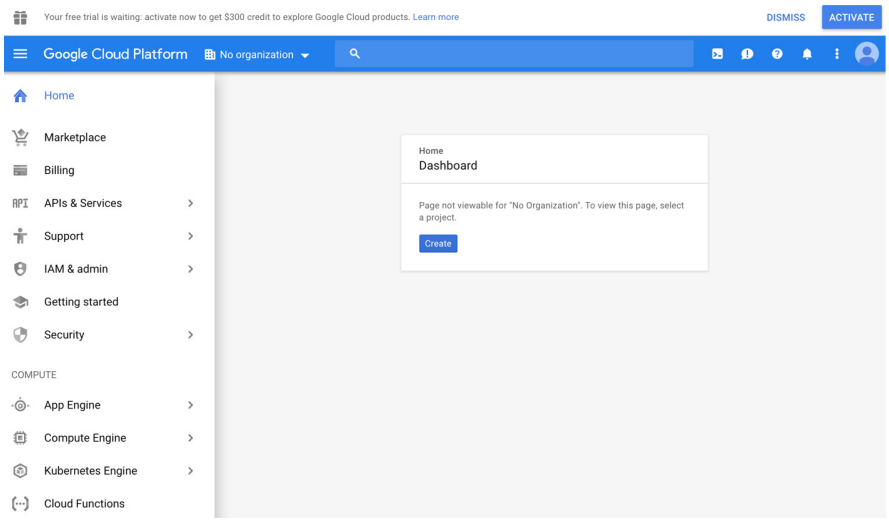

Figure 3.18: Google Cloud Console

2. Click on **Create** from the preceding screenshot to create a new project in Google Cloud for managing resources related to the Kubernetes cluster. You will be taken to the following window:

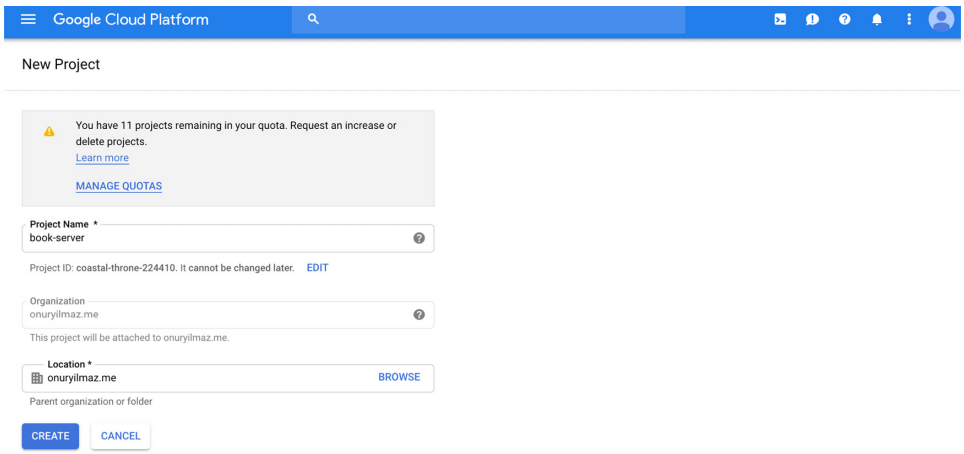

Figure 3.19: Google Cloud project setup

3. Fill in **Project Name** with **`book-server`** and select whether you are part of an organization, such as a company or school.

4. Open the Kubernetes cluster view under the **Kubernetes Engine** menu and wait until Kubernetes Engine is ready as shown below:

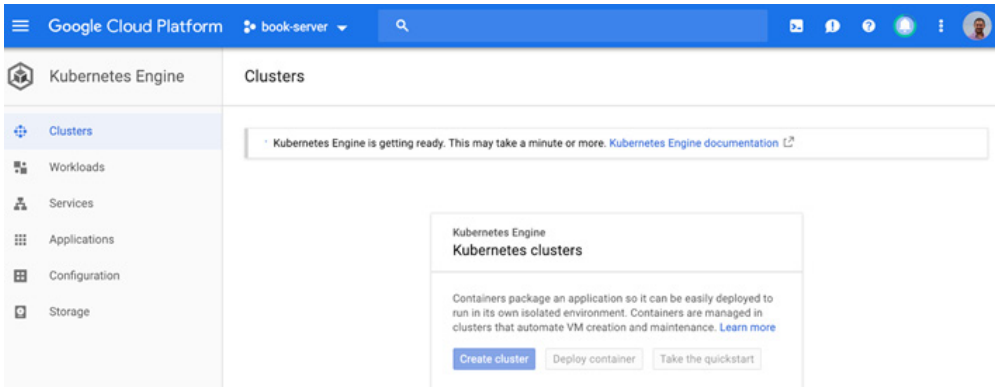

Figure 3.20: The Google Cloud - Kubernetes Engine

Note

When Google Cloud is used for the first time, it could take a couple of minutes to setup user permissions and enable the Kubernetes API. Under the Kubernetes cluster view, you may get the following message: "Kubernetes Engine is getting ready. This may take a minute or more".

5. Create a Kubernetes cluster by opening the GitLab interface, and then in the project section, open Operations. Finally, click Kubernetes in the menu, as shown in the following screenshot:

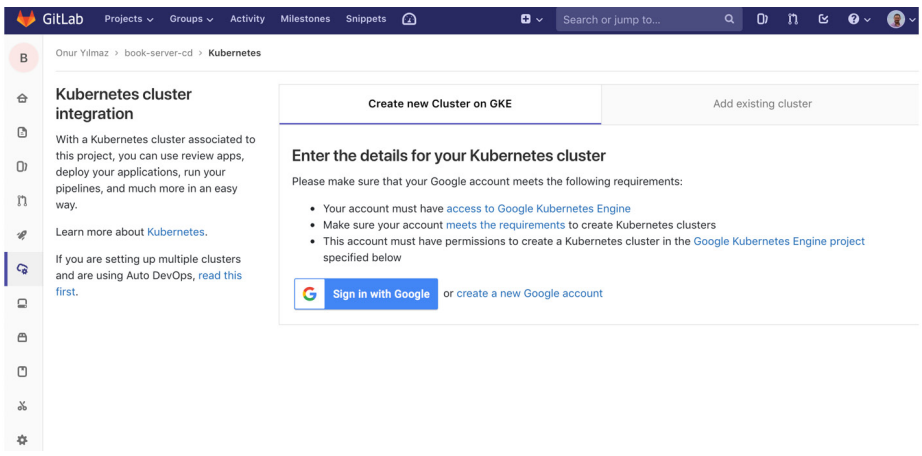

Figure 3.21: The Google Cloud - Kubernetes Engine

6. Click **Add Kubernetes cluster**, and then click **Sign in with Google** under the **Create new Cluster on GKE** tab. After selecting your Google account and providing access, you can create a cluster inside GitLab, as shown

here:

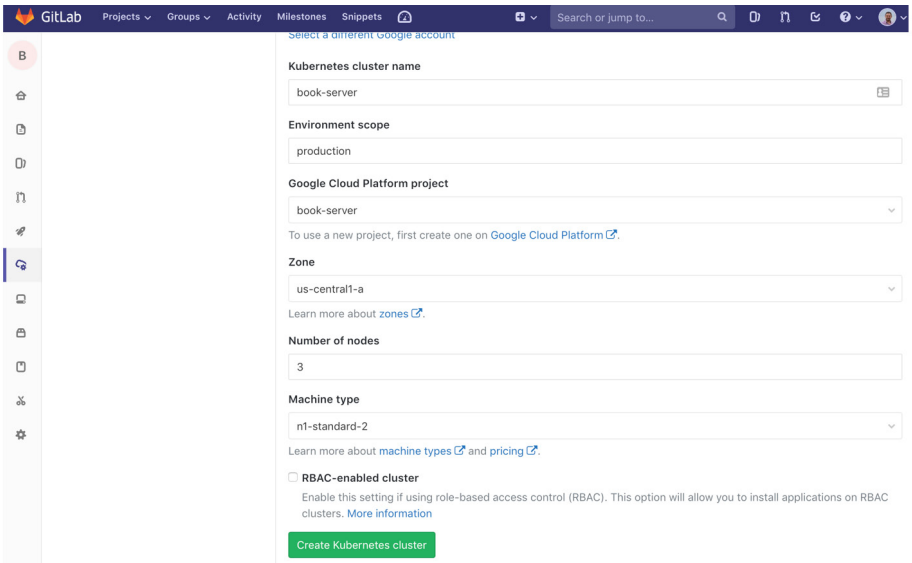

Figure 3.22: Create Kubernetes cluster in GitLab

In this page, please ensure that the cluster name is **`book-server`** and that the **Environment scope** is **`production`**. The scope is used in CI/CD pipelines to deploy applications. In addition, ensure that the **Google Cloud Platform project** is correct and that the **Number of nodes** is 3.

Note

It is also possible to create a Kubernetes cluster in the Google Cloud dashboard and connect to Gitlab, but various authentication-related fields need to be filled in.

7. Open the Google Cloud Dashboard, open the **Kubernetes** cluster view under the **Computer - Kubernetes Engine** menu, and wait until the cluster is green:

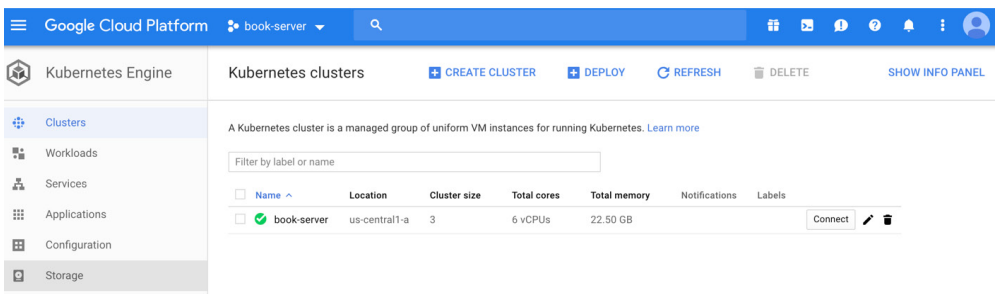

Figure 3.23: Kubernetes Cluster list view

8. Click **Connect** and copy the command shown in **Command-line access** for local usage, as shown in the following screenshot:

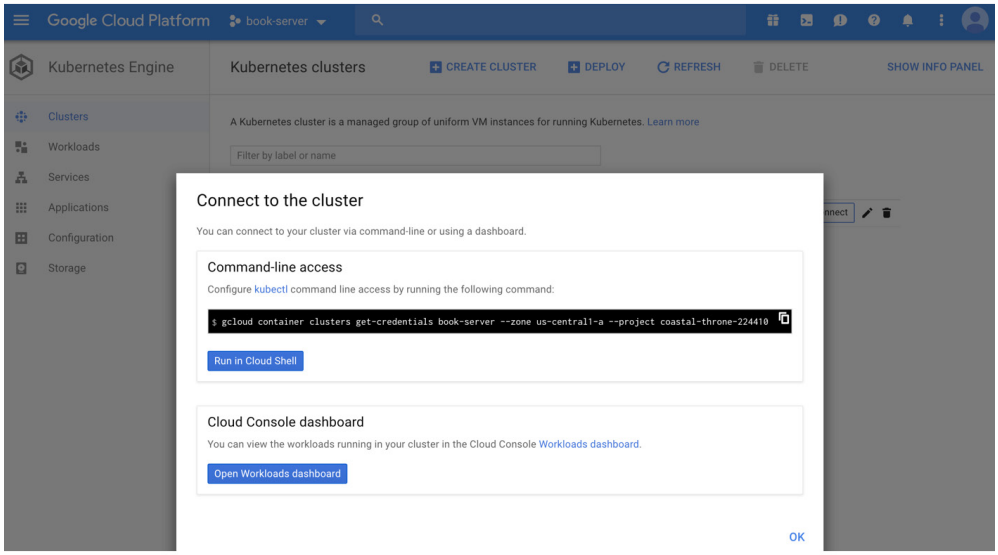

Figure 3.24: Connecting to the cluster

9. Download gcloud to connect Google Cloud locally by running the following command on the shell:

   curl https://sdk.cloud.google.com | bash

10. Initialize `gcloud` with your Google account and the correct project after restarting your terminal:

    gcloud init

    Note

    Further information about Google Cloud SDK is available in the official documentation: https://cloud.google.com/sdk/docs/downloads-interactive.

11. Install the Kubernetes client tool `kubectl` locally by downloading the binary for your operating system:

    # macOS

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl

# Linux

curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl

# Windows

curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.12.0/bin/windows/amd64/kubectl.exe
```

12. Add the binary to the **`PATH`** for command-line access:

```
chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin/kubectl
```

Note

Further information about **`kubectl`** *is available in the official documentation: https://kubernetes.io/docs/tasks/tools/install-kubectl/.*

13. Configure **`kubectl`** for connecting to the Kubernetes cluster by running the following command on the shell (this command has been copied from step 7):

```
gcloud container clusters get-credentials book-server --zone us-central1-a --project coastal-throne-224410
```

Note

This command is specific to the Google Cloud project and you should have details specific to your project.

The following output is obtained:

```
 /cloud-native $ gcloud container clusters get-credentials book-server --zone us-central1-a --project coastal-throne-224410
Fetching cluster endpoint and auth data.
kubeconfig entry generated for book-server.
 /cloud-native $
```

Figure 3.25: Credentials for the GKE cluster

The output shows that **`"kubeconfig entry generated for book-server"`**, which shows that **`kubectl`** is ready to use for the **`book-server`** cluster.

14. Check the nodes in the cluster to test that **`kubectl`** is working by running the following command on the shell:

    kubectl get nodes

    The following output is obtained:

```
 /cloud-native $ kubectl get nodes
NAME                                          STATUS    ROLES     AGE       VERSION
gke-book-server-default-pool-2fb9e85d-gds5    Ready     <none>    3m        v1.9.7-gke.11
gke-book-server-default-pool-2fb9e85d-r92s    Ready     <none>    3m        v1.9.7-gke.11
gke-book-server-default-pool-2fb9e85d-x1xl    Ready     <none>    3m        v1.9.7-gke.11
 /cloud-native $
```

Figure 3.26: Kubernetes cluster nodes

You should see three nodes, as defined in step 3, while creating the cluster.

We have created a Kubernetes cluster to run our cloud-native scalable application in the Google Cloud platform. Next, we will discuss how to package the configuration of Kubernetes applications using Helm.

HELM

Installing applications to Kubernetes includes defining

interdependent resources such as **`volume`** for persistence, **`configmap`** for configuration, **`secret`** for passwords, **`pods`** for multiple containers, and maybe a **`service`** to reach pods outside the cluster. All of these resources could be defined as **`JSON`** or **`YAML`** files and can be installed via **`kubectl`** commands. However, changing multiple files gets complicated when there is a need for an upgrade or a configuration change. Although there are various templating methods and scripts, Helm is the accepted and adopted solution for Kubernetes that works as a package manager to deploy complex applications.

Helm packages are named charts, and is where Kubernetes resource templates and configuration values are separated. For the **`book-server`**, we can check the helm folder structure by using the **`tree`** command. You will observe the following:

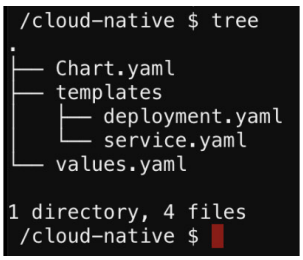

Figure 3.27: Contents of the book-server Helm chart

As can be seen from the preceding screen shot, we can view the following files:

- **`Chart.yaml`** contains metadata about the chart such as name, version, and description.
- **`templates`** contains the template of the Kubernetes resources. One **`deployment`** and one **`service`** is defined in the corresponding YAML files in this folder.
- **`values.yaml`** contains the default configuration values.

Helm works with a server-side backend running in Kubernetes, namely `tiller`, and a command-line client, namely `helm`. It is easy to set up and it integrates well with Kubernetes, since it is the official package manager. Helm also has an active chart repository where popular stable helm charts are maintained (https://github.com/helm/charts/tree/master/stable). Stable charts include various applications that can be installed in Kubernetes such as databases (**MySQL**, **MongoDB**), content management systems (**Joomla** and **Wordpress**), and even game servers (**Minecraft**).

In the following exercise, we will install Helm in the cluster and deploy the book-server application with its Helm chart.

EXERCISE 11: DEPLOYING APPLICATIONS USING HELM

In this exercise, we will install and use `helm` for installing `book-server` into Kubernetes. If you have already installed `helm` locally, you can start from step 2. If you have also installed `tiller`, you can start from step 3:

1. Install the Helm client locally by running the following official script on the shell:

   curl https://raw.githubusercontent.com/helm/helm/master/scripts/get | bash

   You will see the following output:

```
 /cloud-native $ curl https://raw.githubusercontent.com/helm/helm/master/scripts/get | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  7236  100  7236    0     0  37212      0 --:--:-- --:--:-- --:--:-- 37298
Helm v2.11.0 is available. Changing from version v2.12.0-rc.1.
Downloading https://kubernetes-helm.storage.googleapis.com/helm-v2.11.0-darwin-amd64.tar.gz
Preparing to install helm and tiller into /usr/local/bin
helm installed into /usr/local/bin/helm
tiller installed into /usr/local/bin/tiller
Run 'helm init' to configure helm.
 /cloud-native $
```

Figure 3.28: Download and installation of Helm

2. Install Helm on Kubernetes, namely Tiller:

   helm init

```
 /cloud-native $ helm init
$HELM_HOME has been configured at /var/root/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' po
licy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/using_helm/#secu
ring-your-helm-installation
Happy Helming!
 /cloud-native $
```

Figure 3.29: Installation of Tiller

Wait a couple of minutes until **`helm`** is installed to Kubernetes. You can check the available number of **`tiller-deploy`** deployment instances in the **`kube-system`** namespace:

kubectl get deployment tiller-deploy -n kube-system

You will obtain the following output by running the above code.

```
 /cloud-native $ kubectl get deployment tiller-deploy -n kube-system
NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
tiller-deploy   1         1         1            1           6d
 /cloud-native $
```

Figure 3.30: Tiller deployment status

The preceding output shows us that 1 of 1 instances is available for the **`tiller-deploy`** deployment, which indicates that the backend for **`helm`** is running successfully.

3. Deploy a **MySQL** chart to Kubernetes using the **`stable`** repository by running the following code:

   helm install stable/mysql --name mysql --set mysqlRootPassword=password,mysqlUser=mysql,mysqlDatabase=d

   By using the preceding command, we are installing the **`stable/mysql`** chart, and the name of the release will be **`mysql`**. In addition, we will set the root password, user, and database parameters to connect to the database. The following output is obtained:

```
 /cloud-native $ helm install stable/mysql --name mysql  --set mysqlRootPassword=password,mys
qlUser=mysql,mysqlDatabase=default
NAME:   mysql
LAST DEPLOYED: Mon Dec  3 15:04:13 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME   TYPE    DATA  AGE
mysql  Opaque  2     1s

==> v1/ConfigMap
NAME        DATA  AGE
mysql-test  1     1s

==> v1/PersistentVolumeClaim
NAME   STATUS   VOLUME    CAPACITY  ACCESS MODES  STORAGECLASS  AGE
mysql  Pending  standard  1s

==> v1/Service
NAME   TYPE       CLUSTER-IP    EXTERNAL-IP  PORT(S)   AGE
mysql  ClusterIP  10.39.247.83  <none>       3306/TCP  1s

==> v1beta1/Deployment
NAME   DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
mysql  1        1        1           0          1s

==> v1/Pod(related)
NAME                     AGE
mysql-59567844b7-5hr8l  1s


NOTES:
MySQL can be accessed via port 3306 on the following DNS name from within your cluster:
mysql.default.svc.cluster.local

To get your root password run:

    MYSQL_ROOT_PASSWORD=$(kubectl get secret --namespace default mysql -o jsonpath="{.data.my
sql-root-password}" | base64 --decode; echo)

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

    kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash -il

2. Install the mysql client:

    $ apt-get update && apt-get install mysql-client -y

3. Connect using the mysql cli, then provide your password:
    $ mysql -h mysql -p

To connect to your database directly from outside the K8s cluster:
    MYSQL_HOST=127.0.0.1
    MYSQL_PORT=3306

    # Execute the following command to route the connection:
    kubectl port-forward svc/mysql 3306

    mysql -h ${MYSQL_HOST} -P${MYSQL_PORT} -u root -p${MYSQL_ROOT_PASSWORD}


 /cloud-native $
```

Figure 3.31: Installing MySQL using Helm

In this output, first, **`helm`** deployment-related information is provided. Following this, all related Kubernetes resources are listed, starting from **`Secrets`** to

Pods. Finally, some notes are presented, which summarize how to connect to the MySQL database that was just created.

4. Check the Helm releases and actual Kubernetes resources that were created by this release by running the following command on the shell:

   helm ls

   You will see the following output:

```
 /cloud-native $ helm ls
NAME    REVISION        UPDATED                         STATUS          CHART           APP VERSION     NAMESPACE
mysql   1               Mon Dec  3 15:04:13 2018        DEPLOYED        mysql-0.10.2    5.7.14          default
 /cloud-native $
```

Figure 3.32: Helm releases in the cluster

   The preceding output lists all of the releases that have been installed in this cluster. Currently, we only have the **mysql** release in the cluster.

5. Check the Kubernetes resources with the following command:

   kubectl get ConfigMap,PersistentVolumeClaim,Service,Deployment,Pod,Secret

   You will obtain the following output:

```
 /cloud-native $ kubectl get ConfigMap,PersistentVolumeClaim,Service,Deployment,Pod,Secret
NAME           DATA      AGE
cm/mysql-test   1         6m

NAME        STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc/mysql   Bound     pvc-50fab703-f704-11e8-8d54-42010a800169   8Gi        RWO            standard       6m

NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
svc/kubernetes   ClusterIP   10.39.240.1    <none>        443/TCP    2h
svc/mysql        ClusterIP   10.39.247.83   <none>        3306/TCP   6m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/mysql   1         1         1            1           6m

NAME                         READY     STATUS    RESTARTS   AGE
po/mysql-59567844b7-5hr8l    1/1       Running   0          6m

NAME                          TYPE                                  DATA      AGE
secrets/default-token-5fqhf   kubernetes.io/service-account-token   3         2h
secrets/gitlab-token          kubernetes.io/service-account-token   3         2h
secrets/gitlab-token-smzw4    kubernetes.io/service-account-token   3         2h
secrets/mysql                 Opaque                                2         6m
 /cloud-native $
```

Figure 3.33: Kubernetes resources in the cluster

The preceding output shows us that with a single Helm chart, the following resources are being created and work coherently: **`ConfigMap`** for database configuration, **`Secret`** for database passwords, **`PersistentVolumeClaim`** for storing the data in a persistent way, **`Service`** to reach pods from other pods, and **`Deployment`** for running the MySQL database server and a related **`pod`** instance.

6. Check the default values defined in the **`values.yaml`** file in the **`helm`** folder inside the **`book-server-cd`** repository:

   replicaCount: 10

   image:

   repository: registry.gitlab.com/onuryilmaz/book-server-cd

   tag: latest

   database: ""

   This file shows that, by default, ten instances of book-server will be installed from the Docker image of **`registry.gitlab.com/onuryilmaz/book-server-cd:latest`**. However, the **`database`** value is empty, and we need to set it while installing it. With the following command, we are installing the chart in the **`./helm`** folder with the name **`book-server`**, and we are setting the database value according to the password and database name from step 3.

7. Install the **`book-server`** application using Helm by

running the following command in the root folder of **`book-server-cd`**:

helm install --name book-server --set database=mysql://root:password@mysql:3306/default ./helm

You will see the following output:

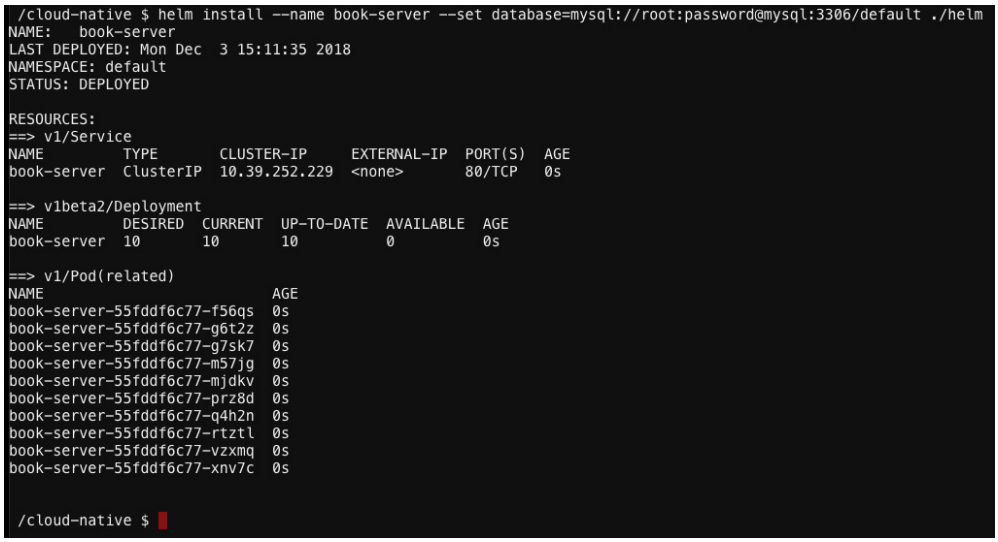

Figure 3.34: Installation of the book-server chart

In this output, first, **`helm`** installation metadata is provided such as name, last deployed time, and status. Then, the Kubernetes resources are listed, which are created due to this **`helm`** installation.

8. Check whether the actual Kubernetes resources were created as expected by running the following command:

   kubectl get Service,Deployment,Pod

   You will obtain the following output:

```
 /cloud-native $ kubectl get Service,Deployment,Pod
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
svc/book-server   ClusterIP   10.39.252.229   <none>        80/TCP     28s
svc/kubernetes    ClusterIP   10.39.240.1     <none>        443/TCP    2h
svc/mysql         ClusterIP   10.39.247.83    <none>        3306/TCP   7m

NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/book-server   10        10        10           10          28s
deploy/mysql         1         1         1            1           7m

NAME                                READY     STATUS    RESTARTS   AGE
po/book-server-55fddf6c77-f56qs     1/1       Running   0          28s
po/book-server-55fddf6c77-g6t2z     1/1       Running   0          28s
po/book-server-55fddf6c77-g7sk7     1/1       Running   0          28s
po/book-server-55fddf6c77-m57jg     1/1       Running   0          28s
po/book-server-55fddf6c77-mjdkv     1/1       Running   0          28s
po/book-server-55fddf6c77-prz8d     1/1       Running   0          28s
po/book-server-55fddf6c77-q4h2n     1/1       Running   0          28s
po/book-server-55fddf6c77-rtztl     1/1       Running   0          28s
po/book-server-55fddf6c77-vzxmq     1/1       Running   0          28s
po/book-server-55fddf6c77-xnv7c     1/1       Running   0          28s
po/mysql-59567844b7-5hr8l           1/1       Running   0          7m
 /cloud-native $
```

Figure 3.35: Kubernetes resources in the cluster

In addition to MySQL-related resources, now, we have one deployment with ten **`pods`** and a **`service`** running to expose our **`book-server`** application. We will check the logs of the **`book-server`** pods in the next step.

9. Check the logs of the **`book-server`** pods by running the following command:

   kubectl logs -l app=book-server

   The following output is obtained:

```
 /cloud-native $ kubectl logs -l app=book-server
time="2018-12-03T14:11:39Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:39Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:38Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:38Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:38Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:38Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:37Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:37Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:41Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:41Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:38Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:38Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:39Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:39Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:41Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:41Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:37Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:37Z" level=info msg="REST server connecting to port 8080"
time="2018-12-03T14:11:40Z" level=info msg="Starting REST server..."
time="2018-12-03T14:11:40Z" level=info msg="REST server connecting to port 8080"
 /cloud-native $
```

Figure 3.36: Logs of book-server instances

As the logs indicate, all 10 **`book-server`** instances are up and running.

10. Connect to the **`book-server`** service inside the cluster and check that its REST API is working by running the following code on the shell:

    kubectl run curl --image=tutum/curl --rm -it

    By using the preceding command, we are running a deployment named **`curl`** with the image of **`tutum/curl`** interactively with the **`-it`** flag; it will be removed when we exit due to the **`--rm`** flag. Within a couple of seconds, this command provides a prompt to allow us to run commands from the local command-line terminal.

11. Check the ping endpoint of the Kubernetes service that we created by running the following code:

    curl book-server/ping

    You will obtain the following output:

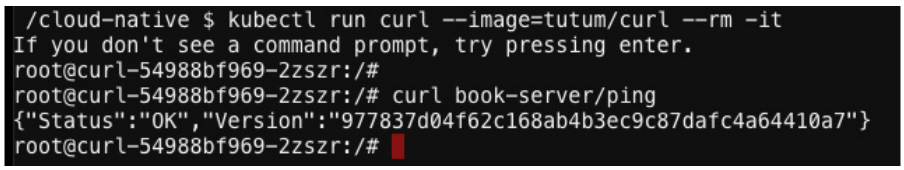

Figure 3.37: Response from the ping endpoint of book-server

    The preceding output shows us that the REST server is running. We can now initialize the **`book-server`** in the next step.

12. Initialize the **`book-server`** by running the following command:

    curl -v book-server/v1/init

    You will get the following output:

```
root@curl-54988bf969-t548b:/# curl -v book-server/v1/init
* Hostname was NOT found in DNS cache
*   Trying 10.39.252.229...
* Connected to book-server (10.39.252.229) port 80 (#0)
> GET /v1/init HTTP/1.1
> User-Agent: curl/7.35.0
> Host: book-server
> Accept: */*
>
< HTTP/1.1 201 Created
< Date: Mon, 03 Dec 2018 14:16:25 GMT
< Content-Length: 0
<
* Connection #0 to host book-server left intact
root@curl-54988bf969-t548b:/#
```

Figure 3.38: Response from the v1/init endpoint of book-server

It is expected that you will see an **`HTTP 201`** response, which actually means **`Created`**, according to HTTP response definitions.

13. Check which books are stored by running the following command:

    curl book-server/v1/books

    You will obtain the following output:

```
root@curl-54988bf969-t548b:/# curl book-server/v1/books
[{"ISBN":"978-1789619270","Title":"Kubernetes Design Patterns and Extensions","Author":"Onur Yilmaz"},{"ISBN":"B07HHDVSJK
","Title":"Cloud-Native Continuous Integration and Delivery","Author":"Onur Yilmaz"}]
root@curl-54988bf969-t548b:/#
```

Figure 3.39: Response from the v1/books endpoint of book-server

As expected, we have retrieved two books from our **`book-server`**. The preceding output shows us that we have successfully installed a database and **`book-server`**, and that they are connected. Besides, we have created ten instances of **`book-server`** and reached them by only using the **`book-server`** address. This shows us that we can scale our application in a cloud-native way with that installation and continue using it with a single address of **`http://book-server`** in the cluster. We also installed the database inside Kubernetes, and it is working in a cloud-native way by provisioning storage from cloud infrastructure.

Helm makes it easy and convenient for installing applications in Kubernetes by creating a layer of abstraction between the Kubernetes resources and the configuration values. This abstraction is valuable to use in CI/CD pipelines, since most of the operations include only changing the values and reinstalling the charts. In the next section, we will discuss cloud-native deployment and update strategies to maintain applications in the cloud.

CLOUD-NATIVE DEPLOYMENT STRATEGIES

Deploying cloud-native applications is straightforward with Helm to Kubernetes; however, there are additional concerns to take care of regarding scalable and reliable applications in the long-run. In this section, we will first discuss the essential characteristics of cloud-native deployments. Then, the most common deployment strategies that satisfy these characteristics will be presented. Following that, we will look at how to implement a deployment strategy using Helm.

The essential characteristics of cloud-native deployments are as follows:

- **Downtime**: While installing or upgrading applications, it is possible to have downtime where no instance of the application is serving. It is critical for the applications that should always be up, like in banking, government, or the defense industry.
- **User Targeting**: For cloud-native applications that serve thousands of users, it is vital to differentiate customers and target them with specific feature sets. It could be their geolocations or device models, or even mobile carriers that marketing or e-commerce applications should focus on.

- **Infrastructure Cost**: The cost of the cloud infrastructure could be high when applications scale to serve large customer bases. Therefore, it is always good to think about infrastructure costs while installing or updating applications.

Deployment strategies are the best practices for deploying the applications in the cloud. The following strategies are the most common and applied strategies that satisfy one or more of the deployment characteristics:

- **Recreate Strategy**: The recreate strategy is based on the idea of removing old versions and then starting the installation of new releases. It is suitable for applications that are flexible on downtime due to the recreate stage. In addition, applications that cannot handle having two different versions running at the same time should implement the recreate strategy.
- **Blue/Green Strategy**: The blue/green strategy is based on the idea of having two installations of the application, namely `blue` and `green`. While the `blue` set serves customers, the `green` set is tested. When the tests pass, the load balancer is switched from `blue` to `green` instances without downtime. This is efficient for applications that do not consider infrastructure costs due to two installations and require high-level production tests.
- **Canary Strategy**: The canary strategy is based on the idea of rolling a new version to a small subset of servers, side by side with the old version. When no problem is being faced, it is possible to roll out to all servers and remove the old version. The idea originates from coal mining, where canary birds are used to alert miners when

the atmosphere is dangerous in the mine. The canaries are used, since they are more sensitive to toxic gases than humans. With the same approach, this strategy is suitable for applications that are tested in production and rolled out when no problem is being faced.

- **A/B Strategy**: The A/B testing strategy is based on the idea of consumer separation and providing different subsets of functionalities, namely A and B groups of customers. It is suitable for applications that could run different versions at the same time, and redirects user based on their characteristics such as language, mobility, or technology.
- **Rolling Update Strategy**: The rolling update strategy is based on the idea of slowly rolling out a version by replacing the previous ones. This strategy is appropriate for the applications that should be always up and running and could handle running two versions of the application at the same time.

With the appropriate deployment strategy and automation, it is possible to deploy and update cloud-native microservices in the cloud reliably. In the following exercise, a rolling update strategy will be applied to `book-server`, and we will demonstrate how the application is upgraded without downtime.

EXERCISE 12: IMPLEMENTING THE ROLLING UPDATE STRATEGY USING HELM

In this exercise, the `book-server` application that we installed in the previous exercise will be upgraded with the rolling update strategy. By default, Kubernetes uses a rolling update strategy for deployment updates. Therefore, we will

use the upgrade capabilities of Helm without further configuration:

1. Connect to the **`book-server`** service inside the cluster and check its version via the REST API:

   kubectl run curl --image=tutum/curl --rm -it

2. Run the following command when the prompt is ready:

   while sleep 1; do curl -s http://book-server/ping; done

   By using the preceding command, the **`ping`** endpoint of the **`book-server`** is called for every second. With the output of this command, we will be able to watch for changes in the version:

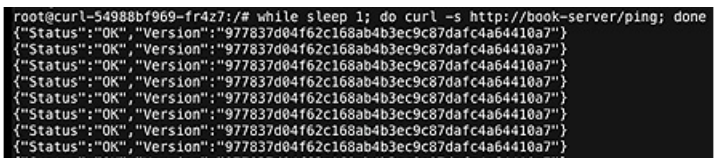

Figure 3.40: ping endpoint response from book-server

   In the preceding output, the response from the **`book-server`** is listed and updated every second. As expected, its status is **`OK`**, and its version is the latest unique tag of the repository.

   Note

   Keep this terminal open since we will follow up the change of versions from this terminal output.

3. Open the GitLab interface and get the commit ID that we can get from the Registry view in Gitlab:

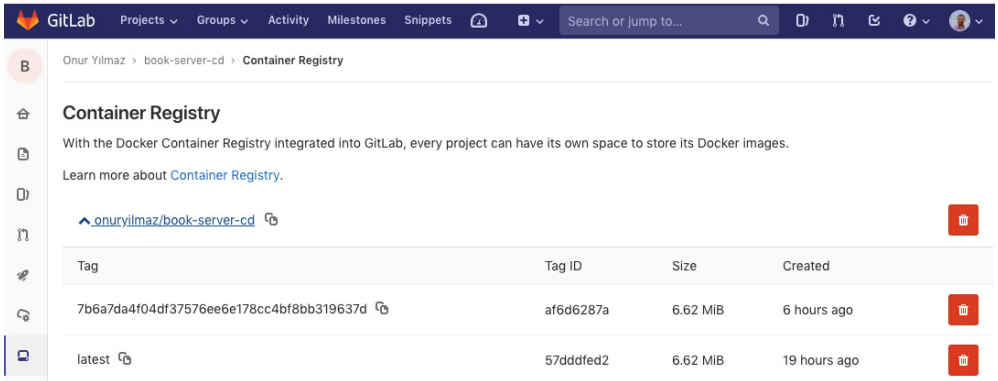

Figure 3.41: Registry with container images in GitLab

4. In another terminal, upgrade the helm release with the commit ID obtained in the previous step using the following command:

   helm upgrade book-server --set image.tag=7b6a7da4f04df37576ee6e178cc4bf8bb319637d ./helm

   You will obtain the following output:

```
 /cloud-native $
 /cloud-native $ helm upgrade book-server --set image.tag=7b6a7da4f04df37576ee6e178cc4bf8bb319637d  ./helm
Release "book-server" has been upgraded. Happy Helming!
LAST DEPLOYED: Mon Dec  3 15:39:55 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME         TYPE       CLUSTER-IP     EXTERNAL-IP  PORT(S)  AGE
book-server  ClusterIP  10.39.252.229  <none>       80/TCP   28m

==> v1beta2/Deployment
NAME         DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
book-server  10       13       5           8          28m

==> v1/Pod(related)
NAME                           AGE
book-server-55fddf6c77-f56qs   28m
book-server-55fddf6c77-g6t2z   28m
book-server-55fddf6c77-g7sk7   28m
book-server-55fddf6c77-m57jg   28m
book-server-55fddf6c77-mjdkv   28m
book-server-55fddf6c77-prz8d   28m
book-server-55fddf6c77-q4h2n   28m
book-server-55fddf6c77-rtztl   28m
book-server-55fddf6c77-vzxmq   28m
book-server-55fddf6c77-xnv7c   28m
book-server-846cc54457-6tml2   0s
book-server-846cc54457-7pfhb   0s
book-server-846cc54457-9g5nx   0s
book-server-846cc54457-jfqht   0s
book-server-846cc54457-vllb7   0s


 /cloud-native $
```

Figure 3.42: Helm output with the upgraded resources

By using the preceding command, we are upgrading the **`book-server`** Helm release by setting the image tag to **`7b6...37d`**. Helm already makes corresponding changes

to Kubernetes resources listed in the preceding output.

5. Check the versions from the terminal that we opened in step 1.

   You will see the following output:

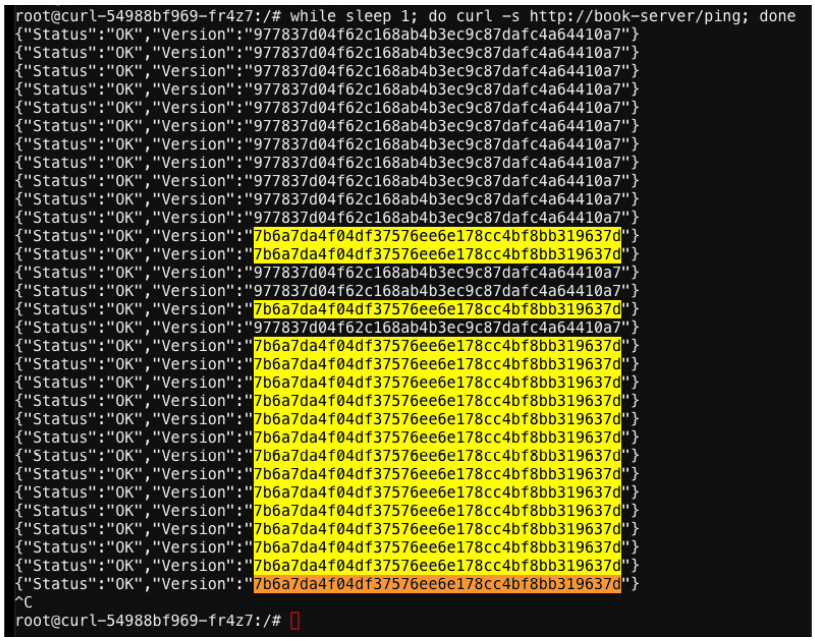

Figure 3.43: Results of the version change in book-server (the new version is highlighted)

We expect to see a rolling upgrade where old versions are removed one by one, and new versions are started where only the new versions are running. The first thing to notice is that none of the requests failed, which indicates no downtime. Secondly, if we see flapping versions between the requests, it demonstrates that both versions are actually running and that the rolling update has not completed yet. Whenever we start to see only new versions, it shows us that the new version has been rolled out and that the old version was eradicated.

In this section, we first introduced Helm and installed the **`book-server`** application using Helm. Then, deployment strategies were discussed, and **`book-server`** was updated without downtime using Helm on Kubernetes. After the first

installation, an appropriate deployment strategy should be selected and used as the last step of the deployment pipeline. With that final step, applications should be automatically updated in cloud platforms without any human interaction.

In the following section, a checklist for a cloud-native delivery/deployment strategy will be presented for a structured transition and implementation.

Checklist for Cloud-Native CD Design

Creating a cloud-native continuous delivery and deployment system requires an in-depth system analysis, well-established planning, and a staged implementation. Similar to the previous chapter, three levels of adoption for CD and deployment are constructed, as follows:

Starter	Intermediate	Advanced
• Manual tag and versioning of containers images • Delivery scripts for container images • Documented manual installation • Deployment environment setup	• Automated tag and versioning of containers images • Automated delivery to container registry • The prototype for a deployment strategy • On-demand deployment to a production system	• Integration into CI pipeline • Automated deployment to production • Alerts and notifications for deployment results

Figure 3.44: A checklist based on different levels of adoption

The Starter stage aims to create container images with manual tagging and delivery by scripts. In addition, the first deployment environment setup should be undertaken. The most common practice in this step is to create some bash scripts for tagging container images. In the Intermediate stage, tagging and versioning of the containers should be automatic, as well as their delivery to registries. Besides, it is

beneficial to create the prototype of a deployment strategy and start on-demand deployment. It is practical to automatically tag the containers with the pipelines in GitLab or other solutions such as AWS CodeBuild, considering the deployment environment. In addition to the rolling update strategy for **`book-server`**, a deployment strategy should be selected, taking into consideration business-critical characteristics. In the last stage, continuous delivery and deployment should be appended to the CI pipeline with the automated deployment to production. With the automated deployment, it is also vital to create alerts and notifications. Within the last stage, all parties in software development should be aware of the pipeline and its business effects, such as live updates, customer support, notifications, and alerts. Like the CI checklist, these levels and starting points for each organization could be different; however, having a structured plan for a CD pipeline transition is usually helpful.

In the following activity, a continuous delivery and deployment pipeline will be designed for the **`book-server`** to version, package, install, and update the application in a cloud-native way.

ACTIVITY 2: BUILDING A CONTINUOUS DELIVERY/DEPLOYMENT PIPELINE FOR CLOUD-NATIVE MICROSERVICES

The aim of this activity is to extend the CI pipeline for the **`book-server`** with the continuous delivery of containers and finally deployment to the Kubernetes cluster. To complete this activity, all the previous exercises across this chapter have to be completed.

The scenario of this activity is as follows: once the production-

ready container has been built at the end of the CI pipeline from the previous chapter, we need the new stages to tag the container and push it to the registry. In addition, only the **`master`** branch should be tagged as the **`latest`** and proceed to installation in production. In other words, a MySQL database and **`book-server`** should be installed/updated using Helm for only the **`master`** branch. The pipeline stages and their statuses should be checked from the GitLab web interface, as shown in the following screenshot:

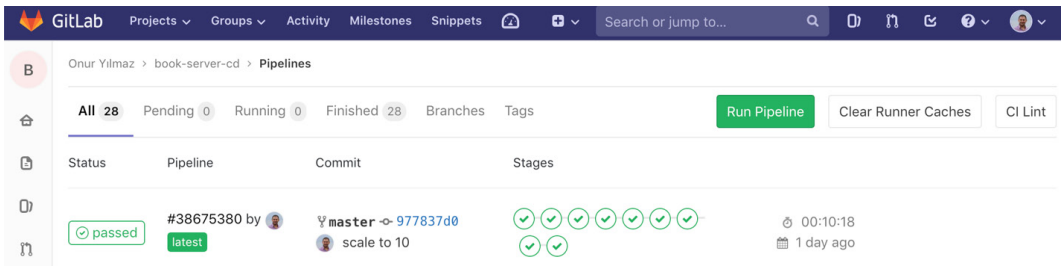

Figure 3.45: CI/CD Pipelines view in GitLab

You should ensure that all of the stages of the pipeline are green and that they are appended to the last stage of the CI pipeline in a sequential way, as shown in the following screenshot:

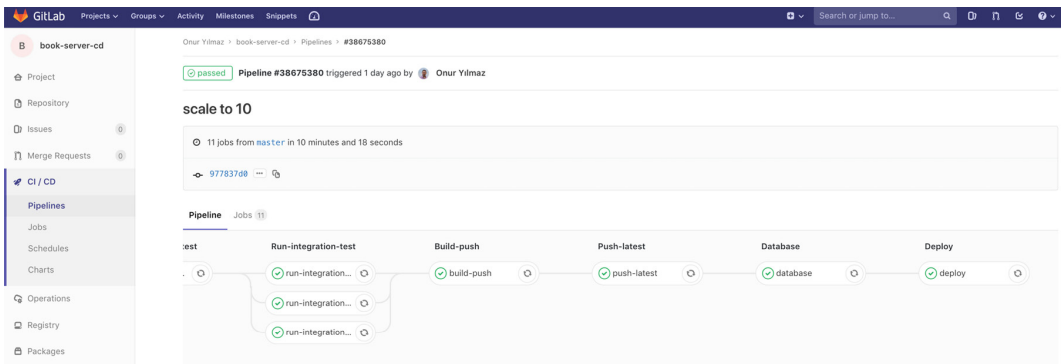

Figure 3.46: Pipeline stages on GitLab

Additionally, with every successful run of the pipeline in the master branch, the **`book-server`** in the Kubernetes cluster should be updated. This can be checked by the **`kubectl describe deployment`** command. Make sure that the

image tag matches the latest command in **`master`**:

kubectl describe deployment book-server

By running the above command, you will see the following output:

```
 /cloud-native $ kubectl describe deployment book-server
Name:                   book-server
Namespace:              default
CreationTimestamp:      Mon, 03 Dec 2018 15:11:35 +0100
Labels:                 app=book-server
Annotations:            deployment.kubernetes.io/revision=10
Selector:               app=book-server
Replicas:               10 desired | 10 updated | 10 total | 10 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=book-server
  Containers:
   book-server:
    Image:      registry.gitlab.com/onuryilmaz/book-server-cd:7b6a7da4f04df37576ee6e178cc4bf8bb319637d
    Port:       8080/TCP
    Liveness:   http-get http://:http/ping delay=0s timeout=1s period=10s #success=1 #failure=3
    Readiness:  http-get http://:http/ping delay=0s timeout=1s period=10s #success=1 #failure=3
    Environment:
      DATABASE:
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   book-server-846cc54457 (10/10 replicas created)
Events:          <none>
 /cloud-native $
```

Figure 3.47: Image of the book-server deployment

Image tags of the container starting with **`7b6a...`** should match the latest commit ID in your repository, which shows that the pipeline successfully updates the deployment in the cluster with the latest commit.

Execute the following steps to complete this activity:

1. Download the forked repository (this was forked in step 1 of exercise 3) to your local system, copy the **`.gitlab-ci.yml`** (https://gitlab.com/TrainingByPackt/book-server-cd/blob/master/.gitlab-ci.yml) definition from the previous chapter where the CI pipeline was completed, and replace the existing **`.gitlab-ci.yml`** file.
2. Create a **`build-push`** stage using the **`docker build`** and **`push`** commands, and use the **`$CI_COMMIT_SHA`** as

commit ID by typing the following code into the `.gitlab-ci.yml` file.

3. Create a `push-latest` stage for the `master` branch to tag the container with the `latest` tag and push it to the registry again.
4. Create a `database` stage using the `devth/helm` image and use the `upgrade` option of `helm`.
5. Create a `deploy` stage similar to the `database` stage for installing `book-server`.
6. Commit the `.gitlab-ci.yml` file to the repository.
7. Open the GitLab interface, click the **CI/CD** tab, and then click the **Run Pipeline** tab.
8. Click **Create pipeline** and then observe the status of the pipeline.

Note

The solution to this activity can be found on page 126.

Note

Please ensure that you remove the Kubernetes cluster created in Exercise 10 if you are not planning to use it for further projects, since keeping it running will cost you money. In order to delete, open the Google Cloud Dashboard and then open Kubernetes cluster view under the menu Computer - Kubernetes Engine and click recycle bin button for the book-server cluster.

Summary

In this third chapter of cloud-native continuous integration

and delivery, the last steps of the delivery pipeline were discussed, starting from versioning containers to installing and updating in the cloud.

First, the versioning of the container images was discussed. Two most common schemes, namely semantic and unique tagging, were presented, alongside examples. It is important to use semantic versions for base images and unique tags in production containers in cloud-native application design. After the correct versioning, the delivery of these containers was covered. A cloud-native and natural way of delivering containers by using a registry was explained. Both a self-hosted Docker registry and a secure cloud Docker registry were explained and demonstrated. It is important to use secure registries when third-party access from customers or end users is incorporated.

Second, the deployment of cloud-native applications is explained by cloud platforms, as well as packaging and deployment strategies. Kubernetes, which is the uprising and prominent open source container orchestration system, was presented with its essential characteristics. An example cluster was created and used throughout this chapter. Following that, installing complex applications to Kubernetes using Helm was presented. Helm works as an abstraction layer between Kubernetes resources and configuration values. We showed you how easy it is to deploy a database application and a REST API server to Kubernetes using Helm and connecting them to each other. Finally, deployment strategies and how they satisfy the characteristics of cloud-native deployments was explained. The most adopted strategies were explained and we showed you how they are implemented in real life.

By the end of this chapter, all of the building blocks of a

complete CI/CD pipeline were covered, starting from static code analysis to rolling updates in the Kubernetes cluster. All of these steps ensure that cloud-native applications are tested, built, delivered, packaged, installed, and updated automatically by following the best practices in the industry. Checklists for different stages of implementation were provided for both CI and CD pipelines as guidelines for an organized transition.

With the best practices, examples, and exercises provided for CI/CD pipelines, it is now possible to bridge the gap between developers and customers efficiently and create reliable, robust, and scalable applications. The chapters of this book can be applied to design so that you can plan and implement CI/CD pipelines for cloud-native applications. With the testing practices that have been covered in this chapter, you can comprehensively test microservices using containers. In addition, you can build them automatically and create production-ready container images. These images can be delivered to customers in a secure way and they can be installed according to the cloud-native deployment strategies. These steps are the guidelines you should use to design and implement an end-to-end CI/CD pipeline for cloud-native applications.

Appendix

About

This section is included to assist the students to perform the activities in the book. It includes detailed steps that are to be performed by the students to achieve the objectives of the activities.

Chapter 2: Cloud-Native Continuous Integration

SOLUTION FOR ACTIVITY 1: BUILDING A CI PIPELINE FOR CLOUD-NATIVE MICROSERVICES

You have been tasked with creating a CI pipeline for a **`book-server`** application that is designed to be built and tested with containers.

To complete this activity, all previous exercises across all chapters will need to be completed. Use the **`test`** and **`build`** functions from the previous exercises with the help of GitLab CI/CD pipelines. Once completed, you should have a complete pipeline, starting with static code analysis, passing all tests, and finally building the production-ready container. The pipeline stages and their statuses should be checked in the

GitLab web interface:

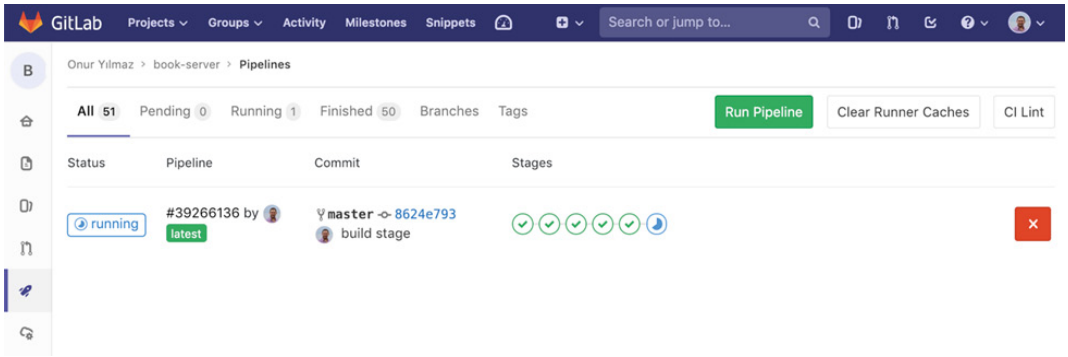

Figure 2.14: CI/CD Pipelines view on GitLab

The pipeline should be green upon successful completion of the activity.

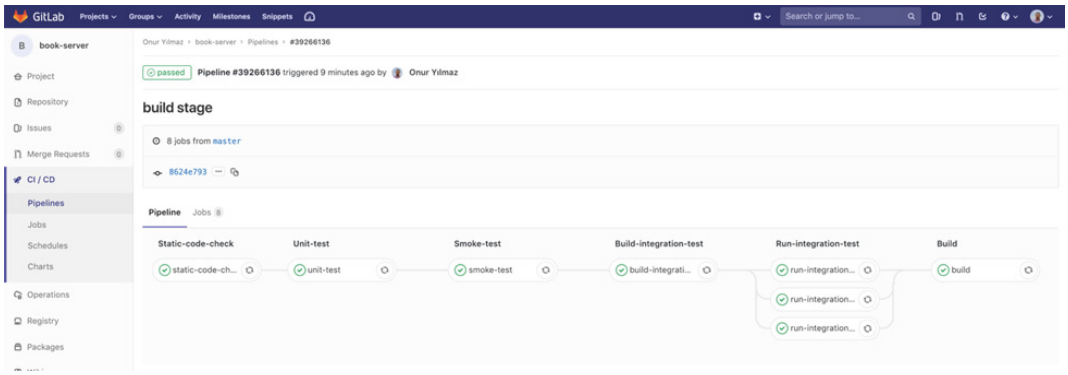

Figure 2.15: Pipeline stages on GitLab

Execute the following steps to complete the exercise:

1. Fork the book-server code to your GitLab project from the **`book-server`** repository (https://gitlab.com/onuryilmaz/book-server) by clicking the **Fork** button:

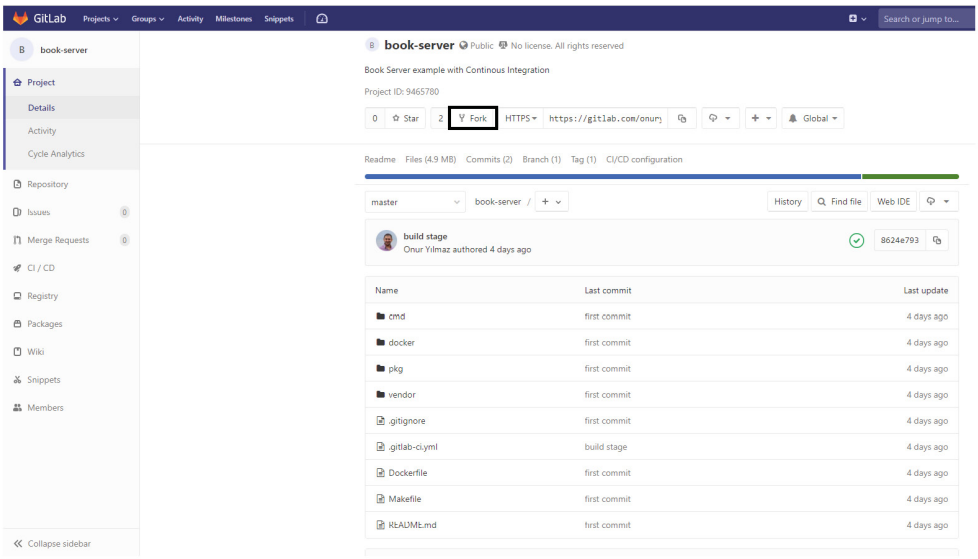

Figure 2.16: Forking the repository on GitLab

2. Delete the existing code in the **`.gitlab-ci.yml`** file. We will be creating all of the stages in this file in the following steps. You should already have the test and build Dockerfiles, along with the source code in the repository once you fork it to your own namespace.

3. Define the stages in the following order in the **`.gitlab-ci.yml`** file to run within Docker-in-Docker, namely **`docker:dind`** service:

   image: docker:latest

   services:

   - docker:dind

   stages:

   - static-code-check

   - unit-test

   - smoke-test

   - build-integration-test

- run-integration-test

- build

4. Create the **`static-code-check`** by typing in the following command in the **`.gitlab-ci.yml`** file:

   static-code-check:

   stage: static-code-check

   script:

   - docker build --rm -f docker/Dockerfile.static-code-check .

5. Create the **`smoke-test`** stage by typing in the following code in the same file, in continuation with the code added in the previous step:

   smoke-test:

   services:

   - docker:dind

   stage: smoke-test

   script:

   - docker build -f docker/Dockerfile.smoke-test -t $CI_REGISTRY_IMAGE/smoke-test:$CI_COMMIT_SHA .

   - docker run -d -p 5432:5432 --name postgres postgres

   - docker run --rm --link postgres:postgres gesellix/wait-for postgres:5432

- docker run -e DATABASE="postgresql://postgres:postgres@postgres:5432/postgr sslmode=disable" --link postgres $CI_REGISTRY_IMAGE/smoke-test:$CI_COMMIT_SHA

6. Create the **`unit-test`** stage by typing in the following code in the same file, in continuation with the code added in the previous step:

   unit-test:

   stage: unit-test

   script:

   - docker build --rm -f docker/Dockerfile.unit-test .

7. Create the **`build-integration-test`** stage by typing in the following code in the same file, in continuation with the code added in the previous step:

   build-integration-test:

   stage: build-integration-test

   script:

   - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY

   - docker build -f docker/Dockerfile.integration-test -t $CI_REGISTRY_IMAGE/integration-test:$CI_COMMIT_SHA .

   - docker push $CI_REGISTRY_IMAGE/integration-test:$CI_COMMIT_SHA

8. Create a **`run-integration`** stage with three parallel jobs for running the tests against MySQL, PostgreSQL, and MSSQL by typing the following code in the same file, in continuation with the code mentioned in the previous step:

   run-integration-test-postgresql:

   services:

   - docker:dind

   stage: run-integration-test

   script:

   - docker run -d -p 5432:5432 --name postgres postgres

   - docker run --rm --link postgres:postgres gesellix/wait-for postgres:5432

   - docker run -e DATABASE="postgresql://postgres:postgres@postgres:5432/postgr sslmode=disable" --link postgres $CI_REGISTRY_IMAGE/integration-test:$CI_COMMIT_SHA

   run-integration-test-mysql:

   services:

   - docker:dind

   stage: run-integration-test

   script:

- docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=password -e MYSQL_DATABASE=default --name mysql mysql

- docker run --rm --link mysql:mysql gesellix/wait-for mysql:3306 -t 30

- docker run -e DATABASE="mysql://root:password@mysql:3306/default" --link mysql $CI_REGISTRY_IMAGE/integration-test:$CI_COMMIT_SHA

run-integration-test-mssql:

services:

- docker:dind

stage: run-integration-test

script:

- docker run -d -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=Password!" -p 1433:1433 --name mssql mcr.microsoft.com/mssql/server:2017-latest

- docker run --rm --link mssql:mssql gesellix/wait-for mssql:1433

- docker run -e DATABASE="mssql://sa:Password!@mssql:1433" --link mssql $CI_REGISTRY_IMAGE/integration-test:$CI_COMMIT_SHA

9. Create a final **`build`** stage to create a production-ready container image by typing in the following code:

build:

stage: build

script:

- docker build --target production -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .

Once done, we will now commit the file to create and run the pipeline.

10. Commit the `.gitlab-ci.yml` file to the repository by running the following commands locally:

    git add .gitlab-ci.yml

    git commit -m "ci pipeline"

    git push origin master

11. Click on the **CI/CD** tab on the GitLab interface and then click on the **Run Pipeline** tab, as shown in the following screenshot:

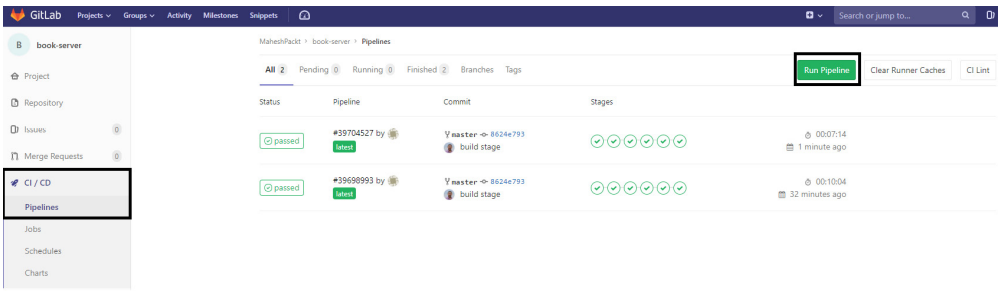

Figure 2.17: Page for running a pipeline

This page will navigate to the following page:

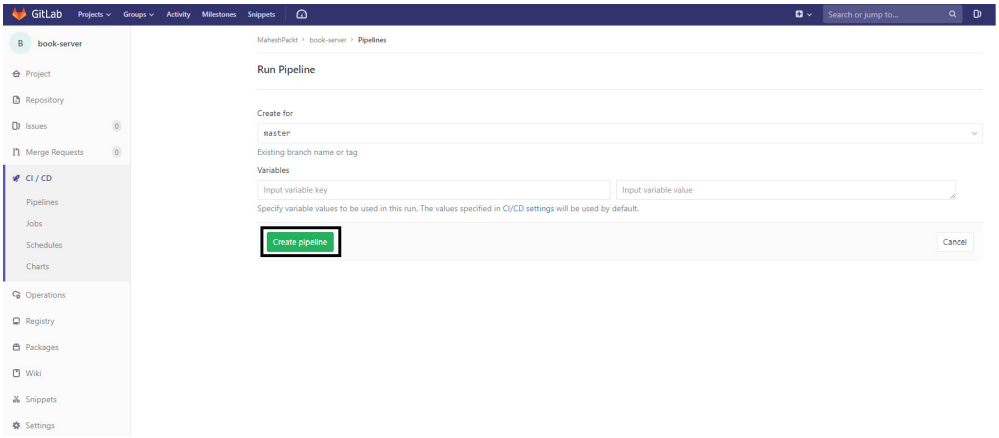

Figure 2.18: Page for creating a pipeline

12. Click on **Create pipeline**, as shown in the previous screenshot. This page navigates to the pipeline view of the jobs. You can see in the following screenshot that all of the stages are presented in the form of a pipeline, with their live status displayed:

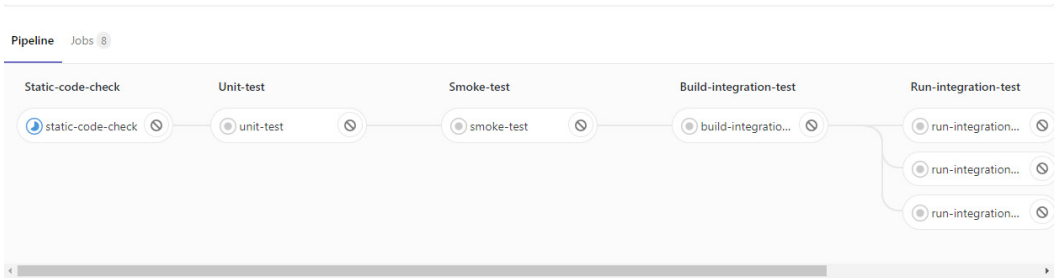

Figure 2.19: Pipeline view of the stages

As each stage is successfully completed, the status is updated, and a green tick is visible, as shown in the following screenshot:

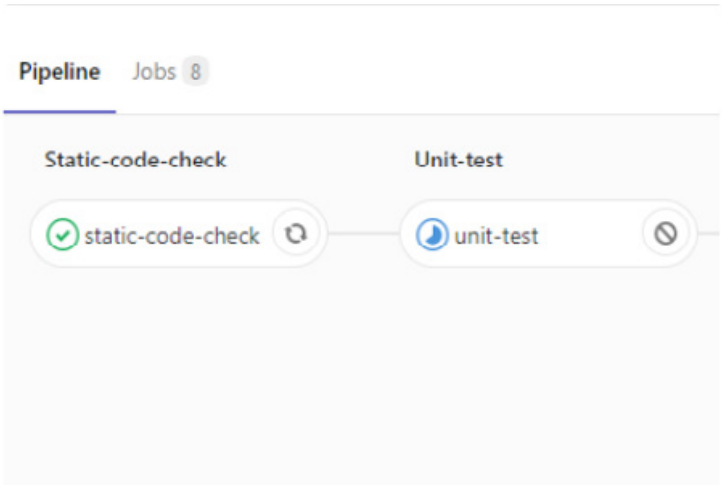

Figure 2.20: Live status update

Once all of the stages are successfully completed, the entire pipeline can be considered to be successfully built, as shown in the following screenshot:

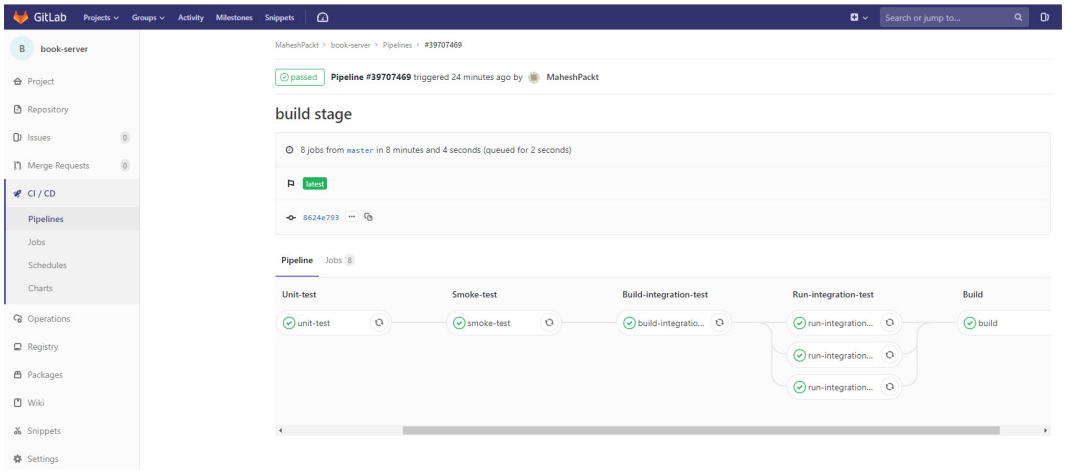

Figure 2.21: A successful pipeline is obtained once all the stages are passed

Note

A pipeline solution is already available in the root folder of book-server in the `.gitlab-ci.yml` *file, which can be found at: https://gitlab.com/TrainingByPackt/book-server/blob/master/.gitlab-ci.yml. The complete code for this activity can be found here: https://bit.ly/2PNhuNV.*

Chapter 3: Cloud-Native Continuous Delivery and Deployment

SOLUTION FOR ACTIVITY 2: CONTINUOUS DELIVERY/DEPLOYMENT PIPELINE FOR CLOUD-NATIVE MICROSERVICES

The aim of this activity is to extend the CI pipeline for the `book-server` with the continuous delivery of containers and finally deployment to the Kubernetes cluster. To complete this

activity, all the previous exercises across this chapter have to be completed.

The scenario of this activity is as follows: once the production-ready container has been built at the end of the CI pipeline from the previous chapter, we need the new stages to tag the container and push it to the registry. In addition, only the **`master`** branch should be tagged as the **`latest`** and proceed to installation in production. In other words, a MySQL database and **`book-server`** should be installed/updated using Helm for only the **`master`** branch. The pipeline stages and their statuses should be checked in the GitLab web interface, as shown in the following screenshot:

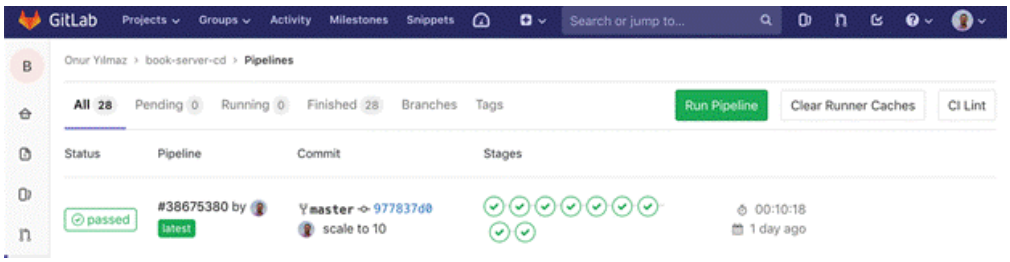

Figure 3.48: CI/CD Pipelines view in GitLab

You should ensure that all of the stages of the pipeline are green and that they are appended to the last stage of the CI pipeline in a sequential way, as shown in the following screenshot:

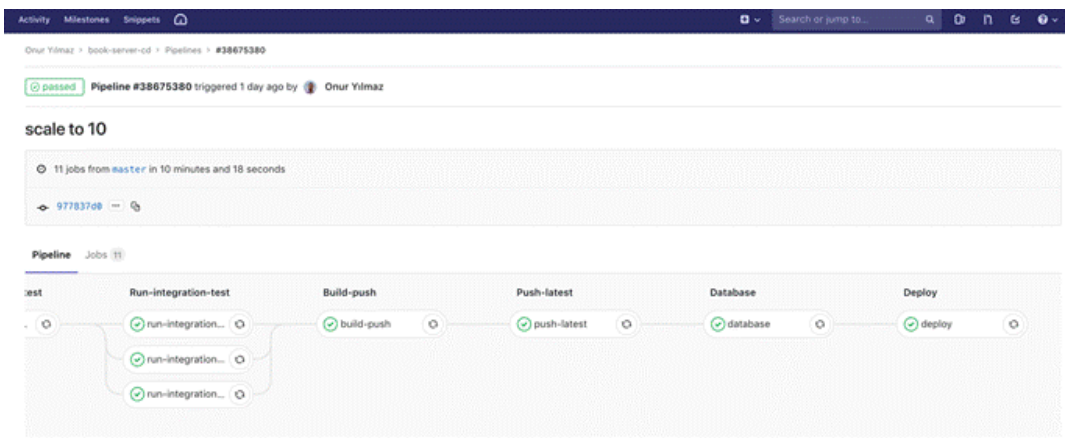

Figure 3.49: Pipeline stages on GitLab

Additionally, with every successful run of the pipeline in the master branch, the **`book-server`** in the Kubernetes cluster should be updated. This can be checked with the **`kubectl describe deployment`** command. Make sure that the image tag matches the latest command in **`master`**:

kubectl describe deployment book-server

By running the above command you should obtain the following output.

```
 /cloud-native $ kubectl describe deployment book-server
Name:                   book-server
Namespace:              default
CreationTimestamp:      Mon, 03 Dec 2018 15:11:35 +0100
Labels:                 app=book-server
Annotations:            deployment.kubernetes.io/revision=10
Selector:               app=book-server
Replicas:               10 desired | 10 updated | 10 total | 10 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=book-server
  Containers:
   book-server:
    Image:      registry.gitlab.com/onuryilmaz/book-server-cd:7b6a7da4f04df37576ee6e178cc4bf8bb319637d
    Port:       8080/TCP
    Liveness:   http-get http://:http/ping delay=0s timeout=1s period=10s #success=1 #failure=3
    Readiness:  http-get http://:http/ping delay=0s timeout=1s period=10s #success=1 #failure=3
    Environment:
      DATABASE:
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   book-server-846cc54457 (10/10 replicas created)
Events:          <none>
 /cloud-native $
```

Figure 3.50: Image of the book-server deployment

Image tags of the container starting with **`7b6a...`** should match the latest commit ID in your repository, which shows that the pipeline successfully updates the deployment in the cluster with the latest commit.

Execute the following steps to successfully complete the activity:

1. Download the forked repository (this was forked in step 1 of exercise 3) to your local system, copy the **`.gitlab-ci.yml`** (https://gitlab.com/TrainingByPackt/book-

server/blob/master/.gitlab-ci.yml) definition from the previous chapter where the CI pipeline where completed, and replace the existing **`.gitlab-ci.yml`** file using the following commands:

git clone https://gitlab.com/<USERNAME>/book-server-cd.git

cd book-server-cd

curl https://gitlab.com/onuryilmaz/book-server/raw/master/.gitlab-ci.yml > .gitlab-ci.yml

With these commands, you will clone the forked repository and then replace the current GitLab pipeline definition with the one from the previous chapter.

2. Create a **`build-push`** stage by using the **`docker build`** and **`push`** commands, using the **`$CI_COMMIT_SHA`** as the commit ID. You can do this by typing the following code into the **`.gitlab-ci.yml`** file:

   build-push:

   stage: build-push

   script:

   - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY

   - docker build --target production --build-arg VERSION=$CI_COMMIT_SHA -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .

   - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA

3. Create a **`push-latest`** stage for the **`master`** branch to tag the container with the **`latest`** tag and push it to the registry again by typing in the following code:

   push-latest:

   stage: push-latest

   only:

   refs:

   - master

   script:

   - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY

   - docker pull $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA

   - docker tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA $CI_REGISTRY_IMAGE:latest

   - docker push $CI_REGISTRY_IMAGE:latest

4. Create a **`database`** stage using the **`devth/helm`** image and use **`upgrade`** option of **`helm`**. You can use the following code to complete this step:

   database:

   stage: database

   image: devth/helm

environment: production

only:

refs:

- master

script:

- helm upgrade --install mysql --set mysqlRootPassword=password,mysqlUser=mysql,mysqlDatabase=d stable/mysql

5. Create a **`deploy`** stage, similar to the **`database`** stage, for installing **`book-server`** by typing in the following code:

   deploy:

   stage: deploy

   image: devth/helm

   environment: production

   only:

   refs:

   - master

   script:

   - helm upgrade --install book-server --set image.repository=$CI_REGISTRY_IMAGE,image.tag=$CI_COMM ./helm

6. Commit the **`.gitlab-ci.yml`** file to the repository by

running the following commands locally:

git add .gitlab-ci.yml

git commit -m "ci pipeline"

git push origin master

7. Open the GitLab interface, click the **CI/CD** tab, and then click the **Run Pipeline** tab, as shown in the following screenshot:

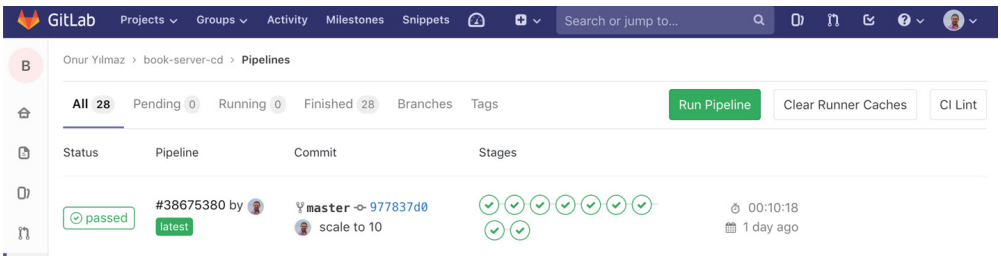

Figure 3.51: Run pipeline page on GitLab

You will navigate to a page with a create pipeline tab, as shown in the following screenshot:

Figure 3.52: Create pipeline page on GitLab

8. Click **Create pipeline** and then observe the status of the pipeline, as shown in the following screenshot:

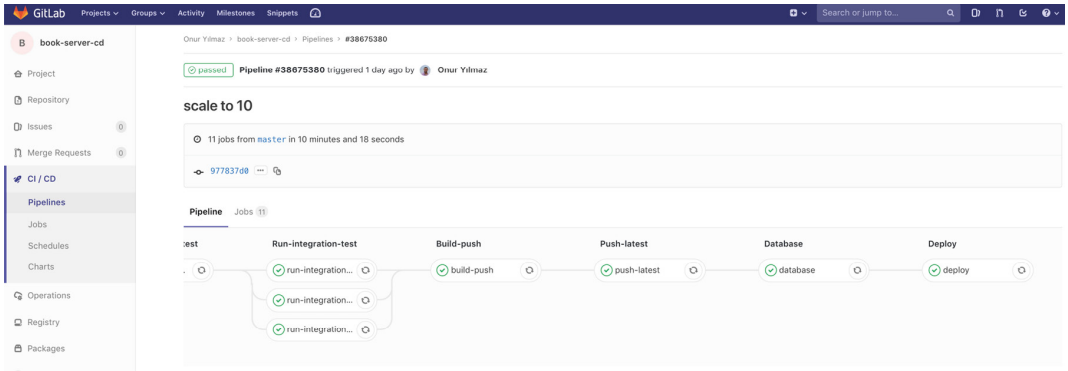

Figure 3.53: A completely successful pipeline

Once all of the stages have been completed successfully, the entire pipeline will turn green, as shown in the preceding screenshot.

Note

A pipeline solution is already available in the root folder of **`book-server-cd`** *in the* **`.gitlab-ci.yml`** *file: https://gitlab.com/TrainingByPackt/book-server-cd/blob/master/.gitlab-ci.yml.*