Hands-On Docker for Microservices with Python

Design, deploy, and operate a complex system with multiple microservices using Docker and Kubernetes

Packt
www.packt.com

Jaime Buelta

Hands-On Docker for Microservices with Python

Design, deploy, and operate a complex system with multiple microservices using Docker and Kubernetes

Jaime Buelta

Packt

BIRMINGHAM - MUMBAI

Hands-On Docker for Microservices with Python

Commissioning Editor: Richa Tripathi
Acquisition Editor: Shriram Shekhar
Content Development Editor: Ruvika Rao
Senior Editor: Afshaan Khan
Technical Editor: Romy Dias
Copy Editor: Safis Editing
Project Coordinator: Francy Puthiry
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Designer: Nilesh Mohite

First published: November 2019

Production reference: 1211119

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83882-381-8

www.packt.com

To my wife, Jordana, for loving me more than I deserve.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Jaime Buelta has been a professional programmer since 2002 and a full-time Python developer since 2010. He has developed software for a variety of fields, focusing, in the last 10 years, on developing web services in Python in the gaming and finance industries. He has seen first hand the revolution of containerization for backend services over the years and has seen how they can improve the development process. He published his first book, *Python Automation Cookbook,* in 2018. He is a strong proponent of automating everything to make computers do most of the heavy lifting so that users can focus on the important stuff. He is currently living in Dublin, Ireland, and is a regular speaker at PyCon Ireland.

This book could not have happened without the encouragement and support of my wonderful wife, Jordana. I also want to thank the team at Packt, especially Manjusha and Ruvika, the editors, who have been a huge help throughout the process. Also, a great thanks to Sean for his great suggestions and for improving the book. Finally, I'd like to thank the whole Python community. I can't overstate what a joy it is to work as a developer in the Python world.

About the reviewer

Sean O'Donnell is a software architect with 19 years of industry experience. He specializes in web applications and systems architecture. Sean has worked on everything from parking meters to Triple A (AAA) games. He is a particular fan of the Python language, ecosystem, and community, and is the founder of Python Ireland. In his spare time, he collects and restores vintage and antique hand tools. Sean lives and works in Dublin, Ireland, with his wonderful fiancé and their three awesome children.

I'd like to thank Aine, Ada, Astrid, and Cian, who put up with me ignoring them while working on this book!

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

The evolution of software means that systems are getting bigger and more complex, making some of the traditional techniques for dealing with them ineffective. The microservice architecture has gained traction in recent years as an effective technique for dealing with complex web services and allowing more people to work on the same system without interfering with one another. In a nutshell, it creates small web services where each one solves a specific problem, and they all coordinate together through well-defined APIs.

In this book, we will explain in detail the microservice architecture and how to successfully run it, enabling you to understand the architecture at a technical level as well as understand the implications of the architecture for teams and their workloads.

For the technical aspects, we will use well-tailored tools, including the following:

- **Python**, to implement RESTful web services
- **Git** source control, to track the changes in an implementation, and **GitHub**, to share those changes
- **Docker** containers, to standardize the operation of each of the microservices
- **Kubernetes**, to coordinate the execution of multiple services
- **Cloud services**, such as Travis CI or AWS, to leverage existing commercial solutions to problems

We will also cover practices and techniques for working effectively in a microservice-oriented environment, the most prominent being the following:

- **Continuous integration**, to ensure that services are of a high quality and ready to be deployed
- **GitOps**, for handling changes in infrastructure
- **Observability** practices, to properly understand what is happening in a live system
- **Practices and techniques aimed at improving teamwork**, both within a single team and across multiple teams

The book revolves around a single example scenario that involves a traditional monolith that needs to be moved to a microservice architecture. This example is described in `Chapter 1`, *Making the Move – Design, Plan, Execute*, and is then continued throughout the rest of the book.

Who this book is for

This book is aimed at developers or software architects who work with complex systems and want to be able to scale the development of their systems.

It is also aimed at developers who typically deal with a monolith that has grown to a point where adding new features is difficult and development is difficult to scale. The book outlines the migration of a traditional monolithic system to a microservice architecture, providing a roadmap covering all the different stages.

What this book covers

`Section 1`, *Introduction to Microservices,* introduces the microservice architecture and the concepts to be used throughout the rest of the book. It also introduces an example scenario that is followed throughout the book.

`Chapter 1`, *Making the Move – Design, Plan, Execute*, explores the differences between the monolith approach and microservices, and how to design and plan a migration from the former to the latter.

`Section 2`, *Designing and Operating a Single Service – Creating Docker Containers*, looks at building and operating a microservice, covering its full life cycle, from design and coding to following good practices to ensure that it's always high quality.

`Chapter 2`, *Creating a REST Service with Python*, covers the implementation of a single web RESTful microservice, using Python and high-quality modules for development speed and quality.

`Chapter 3`, *Build, Run, and Test Your Service Using Docker*, shows you how you can encapsulate a microservice using Docker to create a standard, immutable container.

`Chapter 4`, *Creating a Pipeline and Workflow*, teaches you how to run tests and other operations automatically to ensure that containers are always of high quality and ready to use.

`Section 3`, *Working with Multiple Services: Operating the System through Kubernetes,* moves on to the next stage, which is to coordinate each of the individual microservices so they work as a whole in a consistent Kubernetes cluster.

`Chapter 5`, *Using Kubernetes to Coordinate Microservices*, introduces Kubernetes concepts and objects, including how to install a local cluster.

`Chapter 6`, *Local Development with Kubernetes*, has you deploy and operate your microservices in a local Kubernetes cluster.

`Chapter 7`, *Configuring and Securing the Production System*, delves into the setup and operation of a production Kubernetes' cluster deployed in the AWS Cloud.

`Chapter 8`, *Using GitOps Principles*, describes in detail how to use Git source control to control Kubernetes infrastructure definition.

`Chapter 9`, *Managing Workflows*, explains how to implement a new feature in a microservice, from design and implementation to deployment to an existing Kubernetes cluster system that is open to the world.

`Section 4`, *Production-Ready System: Making It Work in Real-Life Environments,* talks about techniques and tools for the successful operation of a real-life cluster.

`Chapter 10`, *Monitoring Logs and Metrics*, is about monitoring how a live cluster is behaving to proactively detect problems and improvements.

`Chapter 11`, *Handling Change, Dependencies, and Secrets in the System*, is concerned with how to effectively handle configuration that is shared across multiple microservices in a cluster, including the proper management of secret values and dependencies.

`Chapter 12`, *Collaborating and Communicating across Teams,* focuses on the challenges of teamwork between independent teams and how to improve collaboration.

To get the most out of this book

This book uses Python for the code, and assumes that the reader is comfortable reading this programming language, although an expert level is not required.

Git and GitHub are used throughout the book for source control and for tracking changes. It is assumed that the reader is comfortable with using them.

Familiarity with web services and with RESTful APIs is useful for understanding the different concepts that are presented.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781838823818_ColorImages.pdf`.

Code in Action

You can check the code in action videos for this book at `http://bit.ly/34dP0Fm`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "This will generate two files: `key.pem` and `key.pub`, with a private/public pair."

A block of code is set as follows:

```
class ThoughtModel(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50))
    text = db.Column(db.String(250))
    timestamp = db.Column(db.DateTime, server_default=func.now())
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# Create a new thought
new_thought = ThoughtModel(username=username, text=text,
timestamp=datetime.utcnow())
db.session.add(new_thought)
```

Any command-line input or output is written as follows:

```
$ openssl rsa -in key.pem -outform PEM -pubout -out key.pub
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1: Introduction to Microservices

This section covers the first chapter of the book. It introduces the microservice architecture and the problems it aims to solve from a classic monolith system.

`Chapter 1`, *Making the Move – Design, Plan, Execute*, describes a typical situation for a monolith system, its problems, and how the move to microservices can improve the development speed and independent features implementation. It also produces a plan in several steps to facilitate making the move from the initial unique monolith to a multiple RESTful microservice. It also introduces using Docker to implement the different microservices as containers.

In this section, we describe the example system that we will use throughout the book to give a real example of going from a monolithic application to a microservice architecture.

This section comprises the following chapter:

- `Chapter 1`, *Making the Move – Design, Plan, and Execute*

1
Making the Move – Design, Plan, and Execute

As web services get more and more complex, and software service companies grow in size, we require new ways of working to adapt and increase the speed of change, while setting a high quality standard. Microservices architecture has emerged as one of the best tools to control big software systems, enabled by new tools such as containers and orchestrators. We will start by presenting the differences between the traditional monolith architecture and the microservices architecture, as well as the advantages of moving to the latter. We will cover how to structure an architecture migration and how to plan to succeed in this difficult process.

In this book, we will deal with web server services, though some of the ideas can be used for other kinds of software applications, obviously by adapting them. The monolith/microservice architectures have some similarities with the monolithic/microkernel discussions in operating system design, including the famous debate (`https://www.oreilly.com/openbook/opensources/book/appa.html`) between Linus Torvalds and Andrew S. Tanenbaum, back in 1992. This chapter is relatively agnostic about tools, while the following chapters will present specific ones.

The following topics will be covered in this chapter:

- The traditional monolith approach and its problems
- The characteristics of a microservices approach
- Parallel deployment and development speed
- Challenges and red flags
- Analyzing the current system
- Preparing and adapting by measuring usage
- Strategic planning to break the monolith
- Executing the move

At the end of the chapter, you'll be familiar with the basic concepts we will be using throughout the book, different strategies for how to proceed with and structure work during the migration to microservices, and a practical example that we will be working on in the remaining chapters.

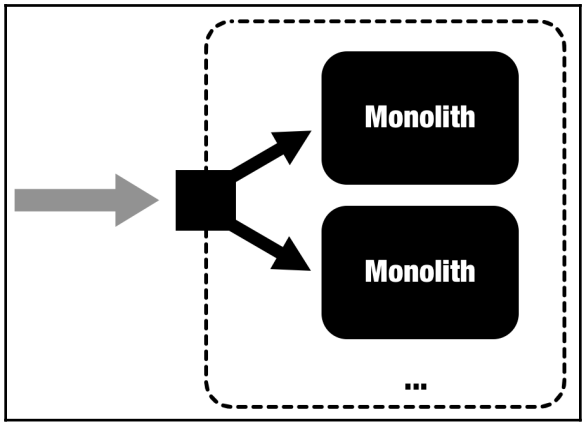
Technical requirements

This chapter does not focus on specific technologies, going for a more agnostic approach. We will discuss a Python Django application for our monolith example.

The monolith example can be found at: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter01/Monolith`. Installation and running instructions can be found in its `README.md` file.

The traditional monolith approach and its problems

The traditional approach to the software when developing a system has been to create a monolith. This is a fancy word to say *a single element, containing everything*, and it is the way virtually every project starts. In the context of web applications, that means creating deployable code that can be replicated so that requests can be directed to any of the deployed copies:



After all, every project will start off small. Making strict divisions early on is inconvenient and even doesn't make sense. A newly created project is small and probably handled by a single developer. While the design can fit in the head of a few people, making strict boundaries between parts of the system is counterproductive.

There are a lot of options for running a web service, but one will typically consist of one or more servers (physical boxes, virtual machines, and cloud instances such as EC2 and more) running a web server application (such as NGINX or Apache) to direct requests directed to HTTP port `80` or HTTPS port `443` toward one or more Python workers (normally, through the WSGI protocol), run by `mod_wsgi`—`https://github.com/GrahamDumpleton/mod_wsgi` (Apache only), uWSGI, GNUnicorn, and so on.

If more than one server is used, there will be a load balancer to spread the load among them. We'll talk about them later in this chapter. The server (or load balancer) needs to be accessible on the internet, so it will have a dedicated DNS and a public IP address.

In other programming languages, the structure will be similar: a frontend web server that exposes the port in HTTP/HTTPS, and a backend that runs the monolith code in a dedicated web worker.

But things change, successful software grows and, after some time, having a big ball of code is maybe not the best way of structuring a big project.

Monoliths can have, in any case, internal structure, meaning they don't necessarily get into the realms of spaghetti code. It may be perfectly structured code. What defines a monolith is the requirement to deploy the system as a whole, without being able to make partial deployments.

Spaghetti code is a common way of referring to code that lacks any structure and is difficult to read and follow.

As the monolith grows, some of its limitations will start to show up:

- **The code will increase in size**: Without strict boundaries between modules, developers will start having problems understanding the whole code base. While good practices can help, the complexity naturally tends to increase, making it more difficult to change the code in certain ways and increasing subtle bugs. Running all tests will become slow, decreasing the speed of any Continuous Integration system.
- **Inefficient utilization of resources**: Each individual deployed web worker will require all the resources required for the whole system to work, for example, the maximum amount of memory for any kind of request, even if a request that demands a lot of memory is rare and just a couple of workers will be sufficient. The same may happen with the CPU. If the monolith connects to a database, each individual worker will require a connection to it, whether that's used regularly or not, and so on.
- **Issues with development scalability**: Even if the system is perfectly designed to be horizontally scalable (unlimited new workers can be added), as the system grows and the development team grows, development will be more and more difficult without stepping on each other's toes. A small team can coordinate easily, but once several teams are working on the same code base, the probability of clashing will increase. Imposing boundaries for teams in terms of ownership and responsibility can also become blurry unless strict discipline is enforced. In any case, teams will need to be actively coordinated, which reduces their independence and speed.
- **Deployment limitations**: The deployment approach will need to be shared across teams, and teams can't be individually responsible for each deployment, as deployment will probably involve work for multiple teams. A deployment problem will bring down the whole system.
- **Interdependency of technologies**: Any new tech needs to fit with the tech in use in the monolith. A new technology, for example, a tool that's perfect for a particular problem, may be complicated to add to the monolith, due to a mismatch of technologies. Updating dependencies can also cause issues. For example, an update to a new version of Python (or a submodule) needs to operate with the whole code base. Some required maintenance tasks, such as a security patch, can cause a problem just because the monolith already uses a specific version of a library, which will break if changed. Adapting to these changes requires extra work too.
- **A bug in a small part of the system can bring down the whole service**: As the service is a whole, any critical issue that affects the stability affects everything, making it difficult to generate quality service strategies or causing degraded results.

As you can see in the examples, most of the monolith issues are growing issues. They are not really important unless the system has a sizeable code base. There are some things that work very well in monoliths, such as the fact that, because there are no boundaries in the code, the code can be changed very quickly and efficiently. But as teams grow and more and more developers are working in the system, boundaries help to define objectives and responsibilities. Too much flexibility becomes a problem in the long term.

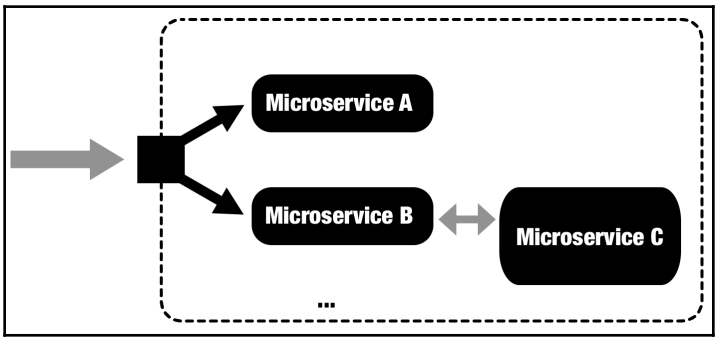
The characteristics of a microservices approach

The monolith approach works until the point it doesn't. But, what is the alternative? That's where the microservices architecture enters into the scene.

A system following a microservices architecture *is a collection of loosely coupled specialized services that work in unison to provide a comprehensive service*. Let's divide the definition a bit, in more specific terms:

1. A **collection of specialized services**, meaning that there are different, well-defined modules.
2. **Loosely coupled**, meaning that each of the microservices can be independently deployed.
3. That **work in unison**—each microservice is capable of communicating with others.
4. To provide a **comprehensive service**, because our microservice system will need to replicate the same functionalities that were available using a monolith approach. There is an intent behind its design.

In contrast to the previous diagram, the microservice architecture will look like this:



Each of the external requests will be channeled to either **Microservice A** or **Microservice B**, each one specializing in a particular kind of requests. In certain cases, **Microservice B** communicates with **Microservice C**, not directly available externally. Note that there may be multiple workers per microservice.

There are several advantages and implications to this architecture:

1. If the communication between microservices is done through a standard protocol, each microservice can be programmed in different languages.

Throughout the book, we will use HTTP requests with data encoded in JSON to communicate between microservices. Though there are more options, this is definitively the most standard and widely-used option, as virtually every widely-used programming language has good support for it.

This is very useful in cases where a specialized language is ideal for a specialized problem, but limiting its use so that it is contained, not requiring a drastic change in the company.

2. Better resource utilization—if **Microservice A** requires more memory, we can reduce the number of worker copies. While on a monolith, each worker requires the maximum resource allocation, now each microservice uses only the resources required for its part of the whole system. Maybe some of them don't need to connect to the database, for example. Each individual element can be tweaked, potentially even at the hardware level.
3. Each individual service is smaller and can be dealt with independently. That means fewer lines of code to maintain, faster builds, and a simpler design, with less technical debt to maintain. There are no dependency issues between services, as each can define and move them at their own pace. Performing refactors can be done in a more controlled way, as they won't affect the totality of the system. Furthermore, each microservice can change the programming language it's written in, without affecting other microservices.

From a certain point of view, the microservices architecture is similar to the UNIX philosophy, applied to web services: write each program (service) to do one thing and do it well, write programs (services) to work together and write programs (services) to handle text streams (HTTP calls), because that is a universal interface.

4. Some services can be hidden from external access. For example, **Microservice C** is only called by other services, not externally. In some scenarios, that can improve security, reducing the attack surface area for sensitive data or services.
5. As the systems are independent, a stability problem in one won't completely stop the system. This reduces critical responses and limits the scope of a catastrophic failure.
6. Each service can be maintained independently by different developers. This allows for parallel development and deployment, increasing the amount of work that can be done by the company. This requires the exposed APIs to be backward compatible, as we will describe later.

Docker containers

The microservice architecture is pretty agnostic about the platform that supports it. It can be deployed on old physical boxes in a dedicated data center, in a public cloud, or in containerized form.

There's a tendency, though, to use containers to deploy microservices. Containers are a packetized bundle of software that encapsulates everything that is required to run, including all dependencies. It only requires a compatible OS kernel to run autonomously.

Docker is the lead actor in containers for web applications. It has an extremely vibrant community supporting it as well as great tooling to work on all kinds of operations. We will learn how to work and operate using Docker.

The first time that I worked with Docker containers, they looked like a sort of *light virtual machine* to me; a small operative system that didn't require simulating the hardware to run. But after a while, I realized that's not the correct approach.

The best way to describe a container is to think of *a process that's surrounded by its own filesystem*. You run one process (or a few related ones), and they *see* a whole filesystem, not shared by anyone.

This makes containers extremely portable, as they are detached from the underlying hardware and the platform that runs them; they are very lightweight, as a minimal amount of data needs to be included, and they are secure, as the exposed attack surface of a container is extremely small. You don't need applications to manage them in the same way you do on a traditional server, such as an `sshd` server, or a configuration tool such as Puppet. They are specialized and designed to be small and single-purpose.

In particular, try to keep your containers small and single-purpose. If you end up adding several daemons and a lot of configuration, it's likely that you are trying to include too much; maybe you need to split it into several containers.

Working with Docker containers has two steps. First, we build the container, executing layer after layer of changes on the filesystem, such as adding the software and configuration files that will be executed. Then, we execute it, launching its main command. We will see exactly how to do this in `Chapter 3`, *Dockerizing the Service*.

The microservices architecture aligns very well with some of the characteristics of Docker containers—small, single-purpose elements that communicate through HTTP calls. That's why, even though it's not a hard requirement, they're typically presented together these days.

The Twelve-Factor App principles (`https://12factor.net/`), which are a collection of practices that have been proven successful in developing web applications, are also very aligned with Docker containers and with the microservice architecture. Some of the principles are extremely easy to follow with Docker, and we will comment on them in depth later in the book.

An important factor for dealing with containers is that containers should be stateless (Factor VI—`https://12factor.net/processes`). Any state needs to be stored in a database and each container stores no persistent data. This is one of the key elements for scalable web servers that, when dealing with a couple of servers, may not be done. Be sure to keep it in mind.

Another advantage of Docker is the availability of a lot of ready-to-use containers. Docker Hub (`https://hub.docker.com/`) is a public registry full of interesting containers to inherit or to use directly, either in development or production. This helps you to have examples for your own services, and to quickly create small services that require little configuration.

Container orchestration and Kubernetes

Though Docker presents on how to deal with each of the individual microservices, we will need an orchestrator to handle the whole cluster of services. For that, we will use Kubernetes (`https://kubernetes.io/`) throughout the book. This is the main orchestration project, and it has great support from the main cloud vendors. We will talk in detail about it in `Chapter 5`, *Using Kubernetes to Coordinate Microservices*.

Parallel deployment and development speed

The single most important element is the capacity to deploy independently. Rule number one for creating a successful microservices system is to ensure that each microservice can operate as **independently** as possible from the rest. That includes development, testing, and deployment.

This is the key element that allows developing in parallel between different teams, allowing them to scale the work. This increases the speed of change in a complex system.

The team responsible for a specific microservice needs to be capable of deploying a new version of the microservice without interrupting any other teams or services. The objective is to increase the number of deployments and the speed of each of them.

The microservice architecture is strongly related to Continuous Integration and Continuous Deployment principles. Small services are easy to keep up to date and to continuously build, as well as to deploy without interruption. In that regard, a CI/CD system tends to be microservices due to the increase in parallelization and the speed of delivery.

As deploying a microservice should be transparent for dependent services, special attention should be paid to backward compatibility. Some changes will need to be escalated and coordinated with other teams to remove old, incorrect functionality without interrupting the system.

While, theoretically, it's possible to have totally disconnected services, that's not realistic in practice. Some services will have dependencies between them. A microservice system will force you to define strong boundaries between the services, and any feature that requires cross-service communication will carry some overhead, maybe even having to coordinate the work across different teams.

When moving to a microservices architecture, the move is not purely technical but also implies a big change in the way the company works. The development of microservices will require autonomy and structured communication, which requires extra effort up front in planning the general architecture of the system. In monolith systems, this may be ad hoc and could have evolved into a not-so-separated internal structure, increasing the risk of tangled code and technical debt.

The need to clearly communicate and define owners cannot be stressed enough. Aim to allow each team to make their own decisions about their code and formalize and maintain the external APIs where other services depend on them.

This extra planning, though, increases long-term delivery bandwidth, as teams are empowered to make more autonomous decisions, including big ones such as which operating system to use, or which programming language, but also a myriad of smaller ones, such as using third-party packages, frameworks, or module structures. This increases the development pace in day-to-day operations.

Microservices may also affect how the teams are structured in your organization. As a general rule, existing teams should be respected. There will be expertise in them that will be very useful, and causing a total revolution will disrupt that. But some tweaks may be necessary. Some concepts, such as understanding web services and RESTful interfaces will need to be present in every microservice, as well as knowledge on how to deploy its own service.

A traditional way of dividing teams is to create an operations team that is in charge of infrastructure and any new deployments because they are the only ones allowed to have access to the production servers. The microservices approach interferes with this as it needs teams to be able to have control over their own deployments. In `Chapter 5`, *Using Kubernetes to Coordinate Microservices*, we'll see how using Kubernetes helps in this situation, detaching the maintenance of the infrastructure from the deployment of services.

It also allows creating a big sense of ownership, as teams are encouraged to work in their own preferred way in their own kingdom, while they play the game with the rest of the teams within clearly defined and structured borders. Microservices architecture can allow experimentation and innovation in small parts of the system that, once proven, can be disseminated across the whole system.

Challenges and red flags

We've discussed a lot of advantages that the microservice architecture has over a monolith, but migrating is a massive undertaking that should not be underestimated.

Systems get started as monoliths, as it is simpler and allows for quicker iteration in a small code base. In any new company, pivoting and changing the code, searching for a successful business model is critical. This takes preference over clear structures and architecture separations—it is the way it should be.

However, once the system matures, the company grows. As more and more developers get involved, the advantages of a monolith start to become less evident, and the need for long-term strategy and structure becomes more important. More structure doesn't necessarily mean moving toward a microservice architecture. A great deal can be achieved with a well-architected monolith.

Moving to microservices also has its own problems. Some of them are the following:

1. Migrating to microservices requires a lot of effort, actively changing the way an organization operates, and a big investment until it starts to pay off. The transition will probably be painful, as a pragmatic approach is required and compromises will need to be made. It will also involve a lot of designing documents and meetings to plan the migration—all while the business continues to operate. This requires full commitment and an understanding of what's involved.
2. Do not underestimate the cultural change—organizations are made of people, and people do not like change. A lot of the changes in microservices are related to different ways of operating and doing things in different ways. While this empowers different teams, it also forces them to clarify their interfaces and APIs and to formalize communication and boundaries. This can lead to frustration and resistance by members of the teams.

There's an adage called Conway's law (`http://www.melconway.com/Home/Conways_Law.html`*) that states that organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. For microservices, this means that divisions between teams should reflect the different services. Having multiple teams working in the same microservice will blur the interfaces. We will discuss Conway's law in detail in* `Chapter 12`*, Collaborating and Communicating across Teams.*

3. There's also a learning curve in learning the tools and procedures. Managing clusters is done differently than a single monolith, and developers will need to understand how to interoperate different services for testing locally. In the same way, that deployment will be different from traditional, local development as well. In particular, learning Docker takes some time to adapt. Plan accordingly and give support and training to everyone involved.

4. Debugging a request that moves across services is more difficult than a monolithic system. Monitoring the life cycle of a request is important and some subtle bugs can be difficult to replicate and fix in development.

5. Splitting a monolith into different services requires careful consideration. A bad division line can make two services tightly coupled, not allowing independent deployment. A red flag in that means almost any change to one service requires a change in the other, even if, normally, it could be done independently. This creates duplication of work, as routinely working on a single feature requires changing and deploying multiple microservices. Microservices can be mutated later and boundaries redefined, but there's a cost associated with that. The same care should be taken when adding new services.
6. There's an overhead in creating microservices, as there's some work that gets replicated on each service. That overhead gets compensated by allowing independent and parallel development. But, to fully take advantage of that, you need numbers. A small development team of up to 10 people can coordinate and handle a monolith very efficiently. It's only when the size grows and independent teams are formed that migrating to microservices starts to make sense. The bigger the company, the more it makes sense.
7. A balance between freedom and allowing each team to make their own decisions and standardize some common elements and decisions is necessary. If teams have too little direction, they'll keep reinventing the wheel over and over. They'll also end up creating knowledge silos where the knowledge in a section of the company is totally nontransferable to another team, making it difficult to learn lessons collectively. Solid communication between teams is required to allow consensus and the reuse of common solutions. Allow controlled experimentation, label it as such, and get the lessons learned across the board so that the rest of the teams benefit. There will be tension between shared and reusable ideas and independent, multiple-implementation ideas.

Be careful when introducing shared code across services. If the code grows, it will make services dependent on each other. This can reduce the independence of the microservices.

8. Following the Agile principles, we know that working software is more important than extensive documentation. However, in microservices, it's important to maximize the usability of each individual microservice to reduce the amount of support between teams. That involves some degree of documentation. The best approach is to create self-documenting services. We'll look at some examples later in the book on how to use tools to allow documenting how to use a service with minimal effort.
9. Each call to another service, such as internal microservices calling each other, can increase the delay of responses, as multiple layers will have to be involved. This can produce latency problems, with external responses taking longer. They will also be affected by the performance and capacity of the internal network connecting the microservices.

A move to microservices should be taken with care and by carefully analyzing its pros and cons. It is possible that it will take years to complete the migration in a mature system. But for a big system, the resulting system will be much more agile and easy to change, allowing you to tackle technical debt effectively and to empower developers to take full ownership and innovate, structuring communication and delivering a high quality, reliable service.

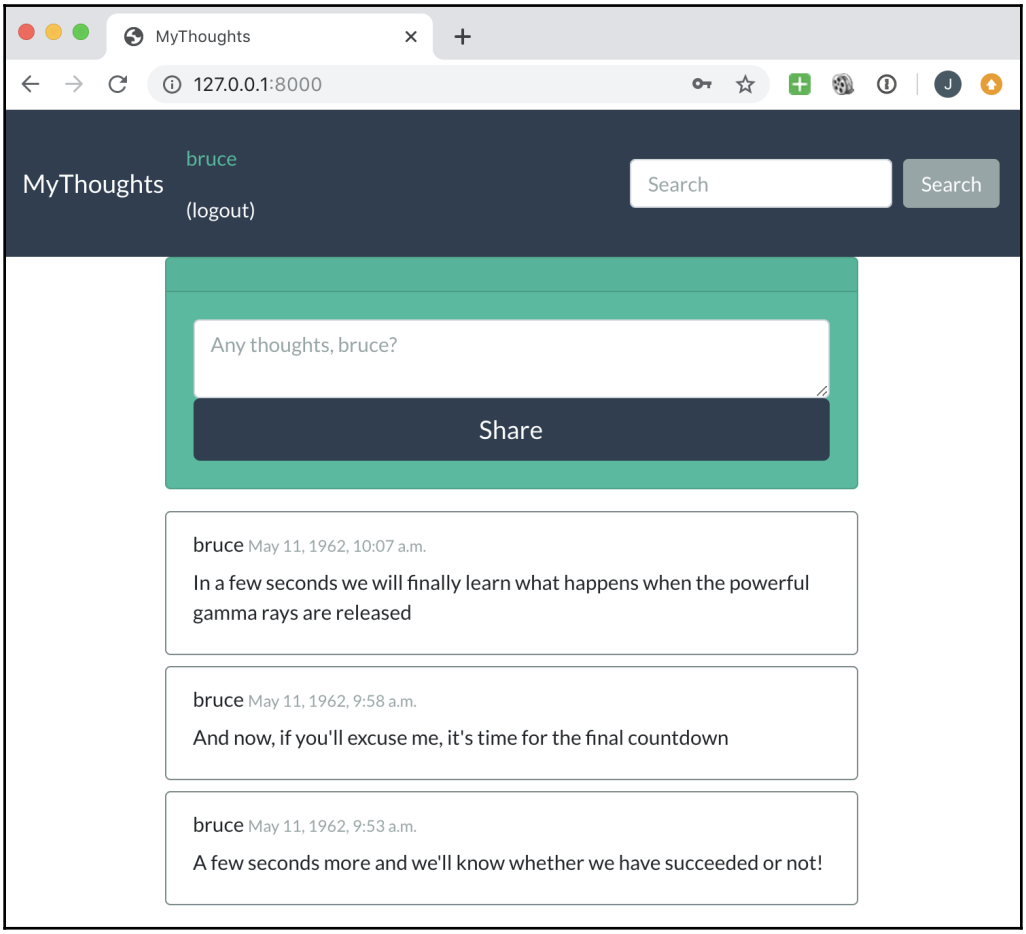
Analyzing the current system

The very first step, as we defined before, to migrate from a monolith to a collection of microservices is understanding the current system. This stage should not be underestimated. It is highly likely that no single person has a good understanding of the different components of the monolith, especially if some parts are legacy.

The objective of this phase is to determine whether a change to microservices will actually be beneficial and to get an initial idea of what microservices will be the result of the migration. As we have discussed, making the move is a big investment and should not be taken lightly. Making a detailed estimation of the effort required won't be possible at this stage; uncertainty will be big at this point, but a thousand-mile journey starts with a single step.

The effort involved will vastly depend on how structured the monolith is. This may vary from a mess of spaghetti code that has grown organically without much direction, to a well-structured and modularized code base.

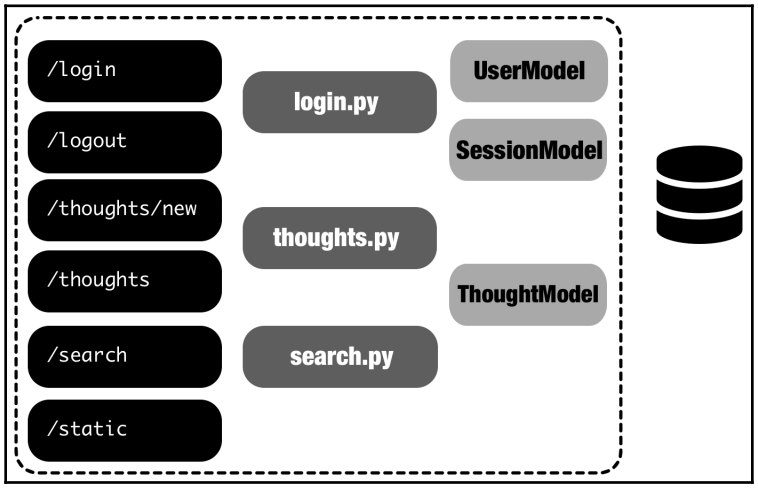
We will use an example application in this book—a micro-blogging site called **MyThoughts**, a simple service that will allow us to post and read short messages or thoughts. The website allows us to log in, post a new thought, see our thoughts, and search for thoughts in the system.



As a first step, we will draw an architectural diagram of the monolith. Reduce the current system to a list of blocks that interact with each other.

The code for our example is available here: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter01/Monolith`. It is a Django application that uses Bootstrap for its HTML interface. See the `README` for instructions on how to run it.

In our example, the MyThoughts model is described in the following diagram:



As you can see, the monolith seems to be following a Model View Controller structure (`https://www.codecademy.com/articles/mvc`):

Django uses a structure called Model Template View, which follows a similar pattern to the MVC one. Read the article at `https://medium.com/shecodeafrica/understanding-the-mvc-pattern-in-django-edda05b9f43f` for more information. Whether it's 100% MCV or not is debatable. Let's not get stuck on semantics, but use the definition as a starting point to describe the system.

- There are three entities stored in a database and accessed through the models: the user, the thoughts, and the session models. The session is used for keeping track of logins.

- A user can log in and out to access the site through the code in `login.py`. If the user logs in, a session is created that allows the user to see the rest of the website.

Please note that the handling of authentication and passwords in this example is for demonstration purposes only. Use the default mechanisms in Django for more secure access. It's the same for the session, where the native session management is not used.

- A user can see their own thoughts. On the same page, there's a new form that creates a new thought. This is handled by the `thoughts.py` file, which retrieves and stores the thoughts through `ThoughtModel`.
- To search other users' thoughts, there's a search bar that connects to the `search.py` module and returns the obtained values.
- The HTML is rendered through the `login.html`, `search.html`, `list_thoughts.html`, and `base.html` templates.
- On top of that, there are static assets that style the website.

This example is very simple, but we are able to see some of the interdependencies:

- The static data is very isolated. It can be changed at any point without requiring any changes anywhere else (as long as the templates are compatible with Bootstrap).
- The search functionality is strongly related to list down thoughts. The template is similar, and the information is displayed in the same way.
- Login and logout don't interact with `ThoughtModel`. They edit the session, but the rest of the application only reads the information there.
- The `base.html` template generates the top bar and it's used for all pages.

After this analysis, some ideas on how to proceed come to mind:

1. Just leave it the way it is, investing in structuring it, but without splitting it into several services. It has a certain structure already, though some parts could be improved. For example, the handling of whether the user is logged in or not could be better. This is obviously a small example, and, in real life, splitting it into microservices would have a big overhead. Remember that sticking with a monolith may be a viable strategy, but if you do, please invest time in cleaning up code and paying technical debt.

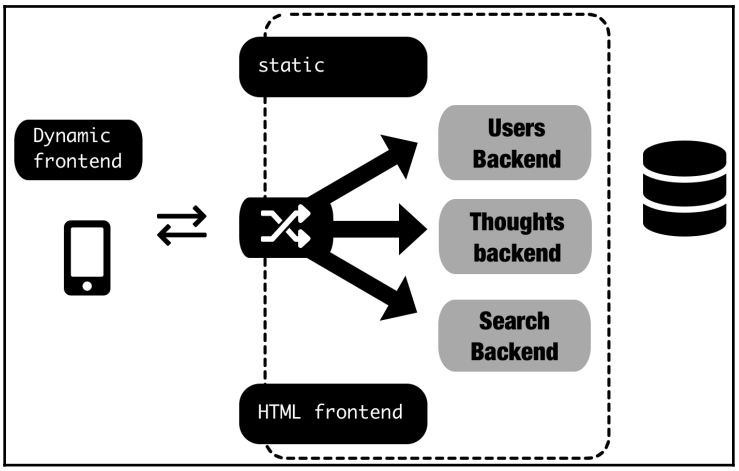
2. Searching for thoughts is pretty basic. At the moment, we directly search the database. If there are millions of thoughts, this won't be a viable option. The code in `search.py` could call a specific search microservice, backed by a search engine such as Solr (`https://lucene.apache.org/solr/`) or Elasticsearch (`https://www.elastic.co/products/elasticsearch`). This will scale the searches and could add capabilities like searching between dates or displaying the text matches. Search is also read-only, so it may be a good idea to detach calls creating new thoughts from calls searching them.
3. Authentication is also a different problem from reading and writing thoughts. Splitting it will allow us to keep on track for new security issues and have a team specifically dealing with those issues. From the point of view of the rest of the application, it only requires you to have something available to check whether a user is logged or not, and that can be delegated in a module or package.
4. The frontend is pretty static at the moment. Maybe we want to create a single-page application that calls a backend API to render the frontend in the client. To do that, a RESTful API microservice that is able to return elements for thoughts and searches will need to be created. The frontend could be coded in a JavaScript framework, such as Angular (`https://angular.io`) or React (`https://reactjs.org/`). In this case, the new microservice will be the frontend, which will be served as static, precompiled code, and will pull from the backend.
5. The RESTful API backend will also be available to allow external developers to create their own tools on top of the MyThoughts data, for example, to create a native phone app.

These are just some ideas, which will need to be discussed and evaluated. What are the specific pain points for your monolithic app? What is the roadmap and the strategic future? What are the most important points and features for the present or the future? Maybe, for one company, having strong security is a priority, and point 3 is critical, but for another, point 5 might be part of the expansion model to work with partners.

The team's structure is also important. Point 4 will require a team with good frontend and JavaScript skills, while point 2 may involve backend optimization and database work to allow an efficient search of millions of records.

Do not jump too quickly to conclusions here; think about what capacity is viable and what your teams can achieve. As we discussed before, the change to microservices requires a certain way of working. Check with the people involved for their feedback and suggestions.

After some consideration, for our example, we propose the following potential architecture:



The system will be divided into the following modules:

1. **Users backend:** This will have the responsibility for all authentication tasks and keep information about the users. It will store its data in the database.
2. **Thoughts backend:** This will create and store *thoughts*.
3. **Search backend**: This will allow searching *thoughts*.
4. A proxy that will route any request to the proper backend. This needs to be externally accessible.
5. **HTML frontend:** This will replicate the current functionality. This will ensure that we work in a backward-compatible way and that the transition can be made smoothly.
6. Allowing clients to access the backends will allow the creation of other clients than our HTML frontend. A dynamic frontend server will be created, and there are talks with an external company to create a mobile app.
7. **Static assets:** A web server capable of handling static files. This will serve the styling for the HTML frontend and the index files and JavaScript files for the dynamic frontend.

This architecture will need to adapt to real-life usage; to validate it, we'll need to measure the existing usage.

Preparing and adapting by measuring usage

Obviously, any real-world system will be more complicated than our example. There's a limit to what a code analysis can discover just by looking at it carefully, and plans often don't survive contact with the real world.

Any division needs to be validated to ensure that it will have the expected result and that the effort will be worth it. So double-check that the system is working the way you think it is working.

The ability to know how a live system is working is called **observability**. The main tools for it are metrics and logs. The problem you'll find is that they will normally be configured to reflect external requests and give no information about internal modules. We will talk about the observability of systems in depth in `Chapter 10`, *Monitoring Logs and Metrics*. You can refer to it for more information and apply the techniques described there at this stage.

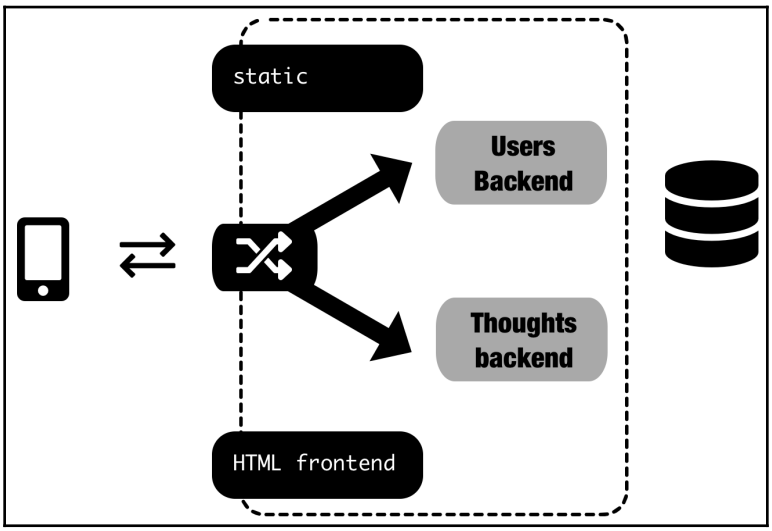
If your system is a web service, by default, it will have activated its access log. This will log each HTTP request that comes into the system and store the URL, result, and time when it happens. Check with your team where these logs are located, as they will provide good information on what URLs are being called.

This analysis, though, will probably give only information about what the external endpoints being called are, but won't say much about internal modules that will be split into different microservices according to our plan. Remember that the most important element for the long-term success of the move to microservices is to allow teams to be independent. If you split across modules that constantly need to be changed in unison, deployments won't be truly independent, and, after the transition, you'll be forced to work with two tightly coupled services.

Be careful, in particular, about making a microservice that's a dependency for every other service. Unless the service is extremely stable, that will make frequent updates likely when any other service requires a new feature.

To verify that the new microservices won't be tightly coupled, make the teams aware of the divisions and how often they have to change the interfaces surrounding them. Monitor these changes for a few weeks to be sure that the division lines are stable and don't require constant change. If the interface between microservices is very actively being changed, any feature will require multiple changes in several services, and that will slow the pace of delivering new features.

In our example, after analyzing the proposed architecture, we decide to simplify the design, as shown in this diagram:



Some changes have been decided after monitoring and talking with the teams:

1. The teams don't have good knowledge of JavaScript dynamic programming. The change to the frontend, at the same time as making the move to microservices, is seen as too risky.
2. The external mobile application, on the other hand, is seen as a strategic move for the company, making the externally accessible API a desirable move.
3. Analyzing the logs, it seems like the search functionality is not often used. The growth in the number of searches is small, and splitting search into its own service will require coordination with the Thoughts Backend, as it's an area of active development, with new fields being added. It is decided to keep search under the Thoughts Backend, as both work with the same thoughts.
4. The Users Backend has been received well. It will allow improving the security of authentication by having clear ownership of who's responsible for patching security vulnerabilities and improving the services. The rest of the microservices will have to work independently with verification by the Users Backend, which means the team responsible for this microservice will need to create and maintain a package with information on how to validate a request.

Once we've decided on the final state, we still have to decide how are we going to move from one state to another.

Strategic planning to break the monolith

As we've discussed previously, moving from the initial state to the desired one will be a slow process. Not only because it involves new ways of doing things, but also because it will happen in parallel with other features and developments that are "business as usual." Being realistic, the company's business activities will not stop. That's why a plan should be in place to allow a smooth transition between one state and the other.

This is known as the **strangler pattern** (`https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler`)—replacing parts of a system gradually until the old system is "strangled" and can be removed safely.

There are a few alternatives as to what technical approach to take to make the move and how to divide each of the elements to migrate to the new system:

- The replacement approach, which replaces the older code with new code written from scratch the new service
- The divide approach, which cherry-picks existing code and moves it into its own new service
- A combination of the two

Let's take a better look at them.

The replacement approach

Services are replaced in big chunks, only taking into account their external interfaces or effects. This black-box approach completely replaces existing functionality coding with an alternative from scratch. Once the new code is ready, it gets activated and the functionality in the old system is deprecated.

Note that this does not refer to a single deployment that replaces the whole system. This can be done partially, chunk by chunk. The basis of this approach is that it creates a new external service that aims to replace the old system.

The pros of this approach are that it greatly helps in structuring the new service, as it doesn't inherit the technical debt, and allows for a fresh look at an old problem, with hindsight.

The new service can also use new tools and doesn't need to continue with any old stack that is not aligned with the strategic views on the future direction of the technology in the company.

The problem with this approach is that it can be costly and can take a long time. For old services that are undocumented, replacing them could take a lot of effort. Also, this approach can only be applied to modules that are stable; if they are developed actively, trying to replace them with something else is moving the goalposts all the time.

This approach makes the most sense for old legacy systems that are small, or at least have a small part that performs limited functionality, and are developed in an old tech stack that's difficult or is no longer considered desirable to maintain.

The divide approach

If the system is well structured, maybe some parts of it can be cleanly split into its own system, maintaining the same code.

In this case, creating a new service is more an exercise of copy-pasting and wrapping it around with the minimal amount of code to allow it to be executed independently and to interoperate with other systems, in other words, to structure its API around HTTP requests to have a standard interface.

If this approach can be used, it means that the code was already quite structured, which is fantastic news.

The systems that are called to this part will also have to be adapted to make the call, not to internal code, but through HTTP calls. The good part is that this can be done in a few steps:

1. Copy the code into its own microservice and deploy it.
2. The old calling system is using the old embedded code.
3. Migrate a call and check that the system is working fine.
4. Iterate until all old calls are migrated to the new system.
5. Delete the divided code from the old system.

If the code is not so cleanly structured, we will need to change it first.

Change and structured approach

If the monolith has been growing organically, it's not likely that all its modules will be cleanly structured. Some structures may exist, but maybe they're not the correct ones for our desired microservices division.

To adapt the service, we will need to make some internal changes. These internal changes could be done iteratively until the service can be cleanly divided.

These three approaches can be combined to generate full migration. The effort involved in each is not the same, as an easily divisible service will be able to make the move faster than a replacement of badly-documented legacy code.

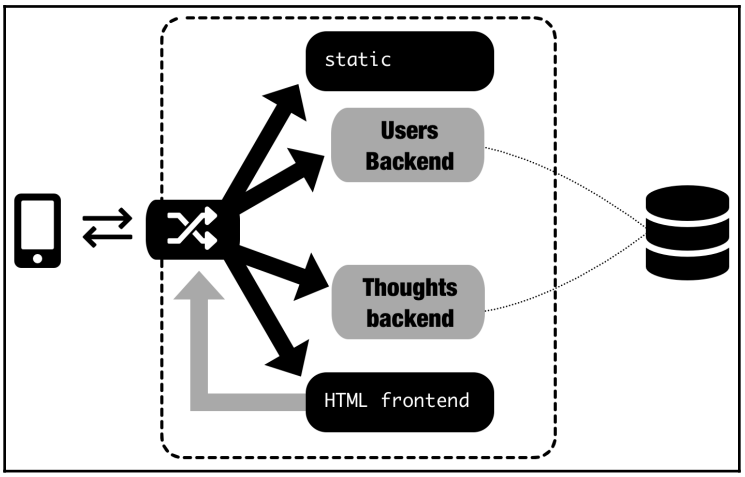
In this phase of the project, the objective is to have a clear roadmap, that should analyze the following elements:

- An ordered plan of what microservices will be available first, taking into account how to deal with dependencies.
- An idea of what the biggest pain points are, and whether working on them is a priority. Pain points are the elements that are worked with frequently and the current way of dealing with the monolith makes them difficult.
- What are the difficult points and the cans of worms? It's likely that there'll be some. Acknowledge that they exist and minimize their impact on other services. Note that they may be the same as the pain points, or not. The difficult points may be old systems that are very stable.
- A couple of quick wins that will keep the momentum of the project going. Show the advantages to your teams and stakeholders quickly! This will also allow everyone to understand the new mode of operation you want to move to and start working that way.
- An idea of the training that teams will require and what the new elements are that you want to introduce. Also, whether there are any skills lacking in your team – it's possible that you may plan to hire.
- Any team changes and ownership of the new services. It's important to consider feedback from the teams, so they can express their concerns over any oversights during the creation of the plan.

For our specific example, the resulting plan will be as follows:

- As a prerequisite, a load balancer will need to be in front of the operation. This will be responsible for channeling requests to the proper microservice. Then, changing the configuration of this element, we will be able to route the requests toward the old monolith or any new microservice.
- After that, the static files will be served through their own independent service, which is an easy change. A static web server is enough, though it will be deployed as an independent microservice. This project will help in understanding the move to Docker.
- The code for authentication will be replicated in a new service. It will use a RESTful API to log in and generate a session, and to log out. The service will be responsible for checking whether a user exists or not, as well as adding them and removing them:
    - The first idea was to check each session retrieved against the service, but, given that checking a session is a very common operation, we decided to generate a package, shared across the externally faced microservices, which will allow checking to see whether a session has been generated with our own service. This will be achieved by signing the session cryptographically and sharing the secret across our services. This module is expected not to change often, as it's a dependency for all the microservices. This makes the session one that does not need to be stored.
    - The Users Backend needs to be able to allow authentication using OAuth 2.0 schema, which will allow other external services, not based on web browsers, to authenticate and operate, for example, a mobile app.
- The Thoughts Backend will also be replicated as a RESTful API. This backend is quite simple at the moment, and it will include the search functionality.
- After both backends are available, the current monolith will be changed, from calling the database directly, to use the RESTful APIs of the backends. After this is successfully done, the old deployment will be replaced with a Docker build and added to the load balancer.
- The new API will be added externally to the load balancer and promoted as externally accessible. The company making the mobile app will then start integrating their clients.

Our new architecture schema is as follows:



Note that the HTML frontend will use the same APIs that are available externally. This will validate that the calls are useful, as we will use them first for our own client.

This action plan can have measurable times and a schedule. Some technology options can be taken as well—in our case, the following:

- Each of the microservices will be deployed in its own Docker container (`https://www.docker.com/`). We will set up a Kubernetes cluster to orchestrate the different services.
- We decided to make the new backend services in Flask (`https://palletsprojects.com/p/flask/`), using Flask-RESTPlus (`https://flask-restplus.readthedocs.io/en/stable/`) to generate a well-documented RESTful app and connect to the existing database with SQLAlchemy (`https://www.sqlalchemy.org/`). These tools are Python, but take a lighter approach than Django.
- The backend services will be served using the uWSGI server (`https://uwsgi-docs.readthedocs.io/en/latest/`).
- The static files will be served using NGINX (`https://www.nginx.com/`).
- NGINX will also be used as a load balancer to control the inputs.
- HTML frontend will continue to use Django (`https://www.djangoproject.com/`).

The teams are OK with proceeding with these tech stacks, and are looking forward to learning some new tricks!

Executing the move

The final step is to execute the carefully devised plan to start moving from the outdated monolith to the new promised land of microservices!

But this stage of the trip can actually be the longest and most difficult—especially if we want to keep the services running and not have outages that interrupt our business.

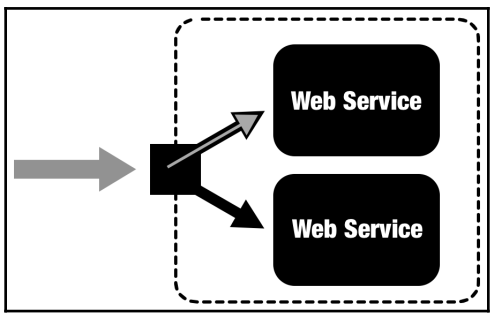
The single most important idea during this phase is **backward compatibility**. This means that the system is still behaving as the old system was from an external point of view. If we are able to behave like that, we can transparently change our internal operation while our customers are able to continue their operations uninterrupted.

That's obviously more easy to say than to do and sometimes has been referred to as starting a race with a Ford T and ending it with a Ferrari, changing every single piece of it without stopping. The good news is that software is so absolutely flexible and malleable that it is actually possible.

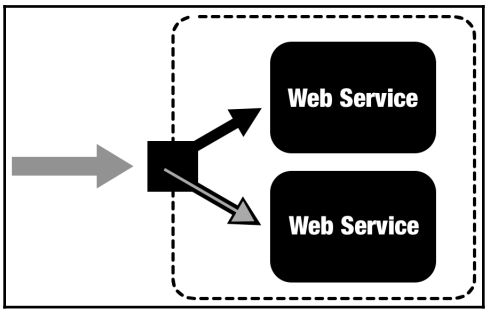
Web services' best friend – the load balancer

A load balancer is a tool that allows distributing HTTP requests (or other kinds of network requests) among several backend resources.

The main operation of a load balancer is to allow traffic to be directed to a single address to be distributed among several identical backend servers that can spread the load and achieve better throughput. Typically, the traffic will be distributed through round-robin, that is, sequentially across all of them:



First one worker, then the other, consecutively:



That's the normal operation. But it can also be used to replace services. The load balancer ensures that each request goes cleanly to one worker or another. The services in the pool of workers can be different, so we can use it to cleanly make the transition between one version of the web service and another.

For our purposes, a group of old web services that are behind a load balancer can add one or more replacement services that are backward compatible, without interrupting the operation. The new service replacing the old one will be added in small numbers (maybe one or two workers) to split the traffic in a reasonable configuration, and ensure that everything is working as expected. After the verification, replace it completely by stopping sending new requests to the old services, draining them, and leaving only the new servers.

If done in a quick movement, like when deploying a new version of a service, this is called a rolling update, so the workers are replaced one by one.

But for migrating from the old monolith to the new microservices, a slower pace is wiser. A service can live for days in a split of 5%/95% so any unexpected error will appear only a twentieth of the time, before moving to 33/66, then 50/50, then 100% migrated.

A highly loaded system with good observability will be able to detect problems very quickly and may only need to wait minutes before proceeding. Most legacy systems will likely not fall into this category, though.

Any web server capable of acting in reverse proxy mode, such as NGINX, is capable of working as a load balancer, but, for this task, probably the most complete option is HAProxy (`http://www.haproxy.org/`).

HAProxy is specialized in acting as a load balancer in situations of high availability and high demand. It's very configurable and accepts traffic from HTTP to lower-level TCP connection if necessary. It also has a fantastic status page that will help to monitor the traffic going through it, as well as taking fast action such as disabling a failing worker.

Cloud providers such as AWS or Google also offer integrated load balancer products. They are very interesting to work from the edge of our network, as their stability makes them great, but they won't be as configurable and easy to integrate into your operating system as HAProxy. For example, the offering by Amazon Web Services is called **Elastic Load Balancing** (**ELB**)—`https://aws.amazon.com/elasticloadbalancing/`.

To migrate from a traditional server with an external IP referenced by DNS and put a load balancer in the front, you need to follow the following procedure:

1. Create a new DNS to access the current system. This will allow you to refer to the old system independently when the transition is done.
2. Deploy your load balancer, configured to serve the traffic to your old system on the old DNS. This way, accessing either the load balancer or the old system, the request will ultimately be delivered in the same place. Create a DNS just for the load balancer, to allow referring specifically to it.
3. Test that sending a request to the load balancer directed to the host of the old DNS works as expected. You can send a request using the following `curl` command:

```
$ curl --header "Host:old-dns.com" http://loadbalancer/path/
```

4. Change the DNS to point to the load balancer IP. Changing DNS registries takes time, as caches will be involved. During that time, no matter where the request is received, it will be processed in the same way. Leave this state for a day or two, to be totally sure that every possible cache is outdated and uses the new IP value.
5. The old IP is no longer in use. The server can (and should) be removed from the externally facing network, leaving only the load balancer to connect. Any request that needs to go to the old server can use its specific new DNS.

Note that a load balancer like HAProxy can work with URL paths, meaning it can direct different paths to different microservices, something extremely useful in the migration from a monolith.

Because a load balancer is a single point of failure, you'll need to load balance your load balancer. The easiest way of doing it is creating several identical copies of HAProxy, as you'd do with any other web service, and adding a cloud provider load balancer on top.

Because HAProxy is so versatile and fast, when properly configured, you can use it as a central point to redirect your requests—in true microservices fashion!

Keeping the balance between new and old

Plans are just plans, and a move to microservices is something to do for internal benefits, as it requires investment until external improvements can be shown in the shape of a better pace of innovation.

This means that there'll be external pressure on the development team to add new features and requirements on top of the normal operation of the company. Even if we make this migration to move faster, there's an initial stage where you'll move slower. After all, changing things is difficult and you will need to overcome the initial inertia.

The migration will take three rough phases.

The pilot phase – setting up the first couple of microservices

A lot of infrastructures may be required before seeing the first deployment. This phase can be difficult to overcome and it's the one that needs the biggest push. A good strategy for that is to put together a dedicated team of **enthusiasts** in the new microservice architecture and allow them to lead the development. They can be people that have been involved in the design, or maybe they like the new technologies or have worked with Docker and Kubernetes on side projects. Not every developer in your team will be excited about changing the way you operate, but some of them will be. Use their passion to start the project and take care of it in its initial steps:

1. Start **small**—there'll be enough work to set up the infrastructure. The objective in this phase is to learn the tools, set up the platform, and adjust how to work with the new system. The aspect of teamwork and coordination is important and starting with a small team allows us to test a couple of approaches and iterate to be sure that they work.
2. Choose **non-critical services**. At this stage, there are a lot of things that can go wrong. Be sure that a problem does not have a huge impact on operations or revenue.
3. Be sure to maintain **backward compatibility**. Substitute parts of the monolith with new services, but do not try to change the behavior at the same time, unless they are obvious bugs.

If there's a new feature that can be implemented as a new microservice, take the chance to go straight for the new approach, but be sure that the risk in extra time to deliver, or bugs, is worth it.

The consolidation phase – steady migration to microservices

After the initial setup, other teams start working with the microservices approach. This expands the number of people dealing with containers and new deployments, so the initial team will need to give them support and training.

Training will be a critical part of the migration project—be sure to allocate enough time. While training events such as workshops and courses can be very useful to kickstart the process, constant support from experienced developers is invaluable. Appoint developers as a point of contact for questions, and tell them explicitly that their job is to ensure that they answer questions and help other developers. Make the supporting team meet up regularly to share concerns and improvements on the knowledge transfer.

Spreading knowledge is one of the main focuses in this phase, but there are another two: clarify and standardize the process and maintain an adequate pace of migrating the microservices.

Documenting standards will be helpful to give clarity and direction. Create checkpoints to make very explicit requirements across the board, so it's very clear when a microservice is ready for production. Create adequate channels for feedback, to be sure that the process can be improved.

During this time, the pace of migration can be increased because a lot of uncertainties and problems have already been ironed out; and because the development will be done in parallel. You should try to work on any new feature in a microservice way, though compromises may need to be taken. Be sure to keep the motivation and follow the plan.

The final phase – the microservices shop

The monolith has been split, and the architecture is now microservices. There may be remains of the monolith that are deemed to have lower priority. Any new feature is implemented in the microservices style.

While desirable, it may not be realistic to migrate absolutely everything from the monolith. Some parts may take a long time to migrate because they are especially difficult to migrate or they deal with strange corners of your company. If that's the case, at least clearly define the boundaries and limit their action radius.

This phase is where the teams can take full ownership of their microservices and start making tests and innovations such as changing the programming language. Architecture can change as well, and microservices can be split or joined. Have clear boundaries defining what the agreed requirements for microservices are, but allow freedom within them.

Teams will be well-established and the process will run smoothly. Keep an eye on good ideas coming from different teams and be sure to spread the word.

Congratulations! You did it!

Summary

In this chapter, we saw what the differences are between the traditional monolith approach and microservices architecture, and how microservices allow us to scale development across several teams and improve the delivery of quality software.

We discussed the main challenges that are faced during a transition from a monolith to microservices and how to perform the change in different stages: analyzing the current system, measuring to validate our assumptions, creating a plan to split the monolith in a controlled way, and tactics to successfully perform the move.

Though this chapter was written in a technology-agnostic way, we've learned why Docker containers are a great way of implementing microservices, something that will be explored in the following chapters. You also now know how using a load balancer helps to maintain backward compatibility and deploy new services in an uninterrupted way.

You learned how to structure a plan to divide a monolith into smaller microservices. We described an example of such a process and an example of a monolith and how it will be divided. We'll see how to do this in detail in the following chapters.

Questions

1. What is a monolith?
2. What are some of the problems of monoliths?
3. Describe the microservice architecture.
4. Which is the most important property of microservices?
5. What are the main challenges to overcome in a migration from a monolith to microservices?
6. What are the basic steps to make such a migration?
7. Describe how to use a load balancer to migrate from an old server to a new one without interrupting the system.

Further reading

You can learn more about systems architecture and how to divide and structure complex systems in the books *Architectural Patterns* (`https://www.packtpub.com/application-development/architectural-patterns`) and *Software Architect's Handbook* (`https://www.packtpub.com/application-development/software-architects-handbook`).

2
Section 2: Designing and Operating a Single Service – Creating a Docker Container

This section follows the creation of a single microservice across three chapters. It starts by presenting an individual REST service implemented in Python, continues with all the requisite steps to implement the service as a self-contained Docker container, and creates a pipeline to ensure that the service always complies with high quality standards.

The first chapter of this section describes the implementation of a single service, following the example presented in the first section. It describes the API interface to be implemented and it uses Python to generate a full-fledged microservice, using tools such as Flask and SQLAlchemy to improve the ease of development. The service includes a testing strategy.

The second chapter of this section shows how to encapsulate the microservice in a Docker container so that the code can be executed immutably through the software life cycle. Basic Docker usage, such as building and running containers, using environment variables, and how to execute testing, is introduced. The process of sharing containers incorporated into a public registry is also described.

The third chapter of this section delves into checking automatically that any new code introduced in the container follows basic quality guidelines, including passing all the tests. It presents continuous integration practices and demonstrates how to create a pipeline in the cloud with Travis CI that will be integrated into a GitHub repository. How to push the resulting container automatically into a registry is also covered in this chapter.

This section comprises the following chapters:

- `Chapter 2`, *Creating a REST Service with Python*
- `Chapter 3`, *Build, Run, and Test Your Service Using Docker*
- `Chapter 4`, *Creating a Pipeline and Workflow*

2
Creating a REST Service with Python

Following our example in the last chapter, we split the system designed as a monolith into smaller services. In this chapter, we will analyze in detail one of the microservices (Thoughts Backend) that we mentioned in the previous chapter.

We will talk about how to develop this microservice as an application using Python. This microservice will be ready to interact with other microservices through a standard web RESTful interface, making it the foundation for our global microservice architecture system.

We will discuss different elements such as the API design, the database schema that supports it, and how to implement and how to implement the microservice. Finally, we'll see how to test the application to be sure that it works correctly.

The following topics will be covered in this chapter:

- Analyzing the Thoughts Backend microservice
- Designing the RESTful API
- Defining the database schema
- Implementing the service
- Testing the code

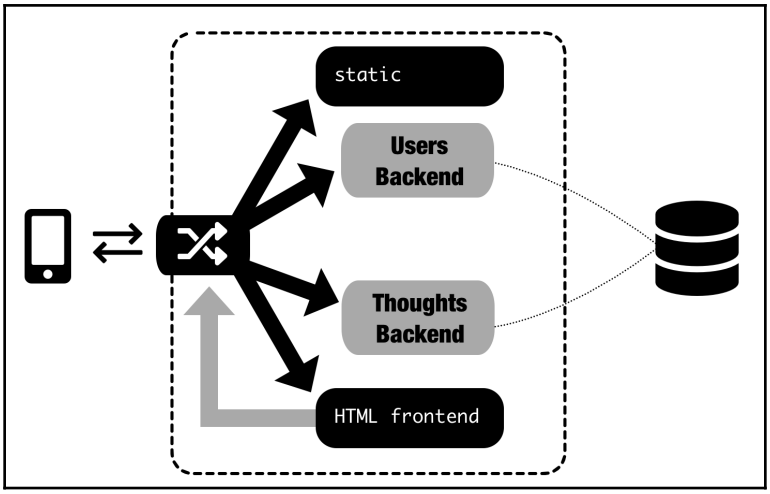
By the end of the chapter, you'll know how to successfully develop a microservice application, including the different stages from design to testing.

Technical requirements

The Thoughts Backend example can be found here (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter02/ThoughtsBackend`). Installation and running instructions can be found on its `README.md` file.

Analyzing the Thoughts Backend microservice

Let's remember the diagram of microservices that we created in the last chapter:



The diagram shows the different elements for our example system: the two backends, users and thoughts, and **HTML frontend**.

Thoughts Backend will be responsible for storing new thoughts, retrieving the existing ones, and searching the database.

Understanding the security layer

As the Thoughts Backend is going to be available externally, we need to implement a security layer. That means we need to identify the user producing the actions and verify their validity. For this service example, we will create a new thought from the logged in user, and we will retrieve my thoughts, thoughts created by the currently logged user.

Note the fact that the user is logged also validates the fact that the user exists.

This security layer will come in the shape of a header. This header will contain information that is signed by the user backend, verifying its origin. It will take the form of a **JSON Web Token** (**JWT**), `https://jwt.io/introduction/`, which is a standard for this purpose.

The JWT itself is encrypted, but the information contained here is mostly only relevant for checking the user that was logged.

A JWT is not the only possibility for the token, and there are other alternatives such as storing the equivalent data in a session cookie or in more secure environments using similar modules such as PASETO (`https://github.com/paragonie/paseto`). Be sure that you review the security implications of your system, which are beyond the scope of this book.

This method should be handled by the **Users Backend** team, and get packaged so that the other microservices can use it. For this chapter, we will include the code in this microservice, but we'll see later how to create it so it's related to the Users Backend.

If the requests don't have a valid header, the API will return a 401 Unauthorized status code.

Note that not all API endpoints require authentication. In particular, `search` does not need to be logged.

With an understanding of how the authentication system is going to work, we can start designing the API interface.

Designing the RESTful API

We will follow the principles of RESTful design for our API. This means we will use constructed URIs that represent resources and then use the HTTP methods to perform actions over these resources.

In this example, we will only use the `GET` (to retrieve), `POST` (to create), and `DELETE` (to delete) methods as the thoughts are not editable. Remember that `PUT` (to overwrite completely) and `PATCH` (to perform a partial update) are also available.

One of the main properties of RESTful APIs is that requests need to be stateless, which means that each request is totally self-contained and can be served by any server. All the required data should be either at the client (that will send it attached to the request) or in a database (so the server will retrieve it in full). This property is a hard requirement when dealing with Docker containers, as they can be destroyed and recreated without warning.

While it is common to have resources that map directly to rows in a database, this is not necessary. The resources can be a composition of different tables, part of them, or even represent something different altogether, such as an aggregation of data, whether certain conditions are met, or a forecast based on analysis on the current data.

Analyze the needs of the service and don't feel constrained by your existing database design. Migrating a microservice is a good opportunity to revisit the old design decisions and to try to improve the general system. Also, remember the Twelve-Factor App principles (`https://12factor.net/`) for improving the design.

It's always good to have a brief reminder about REST before starting an API design, so you can check `https://restfulapi.net/` for a recap.

Specifying the API endpoints

Our API interface will be as follows:

	Endpoint	Requires authentication	Returns
`GET`	`/api/me/thoughts/`	Yes	List of thoughts of the user
`POST`	`/api/me/thoughts/`	Yes	The newly created thought
`GET`	`/api/thoughts/`	No	List of all thoughts
`GET`	`/api/thoughts/X/`	No	The thought with ID `X`
`GET`	`/api/thoughts/?search=X`	No	Searches all the thoughts that contain `X`
`DELETE`	`/admin/thoughts/X/`	No	Deletes thought with ID `X`

Note there are two elements of the API:

- A public API, starting with `/api`:
    - An authenticated public API, starting with `/api/me`. The user needs to be authenticated to perform these actions. A non-authenticated request will return a 401 Unauthorized status code.
    - A non-authenticated public API, starting with `/api`. Any user, even not authenticated, can perform these actions.
- An admin API (starting with `/admin`). This won't be exposed publicly. It spares the authentication and allows you to do operations that are not designed to be done by customers. Clearly labeling with a prefix helps to audit the operations and clearly signifies that they should not be available outside of your data center.

The format of a thought is as follows:

```
thought
{
    id integer
    username string
    text string
    timestamp string($date-time)
}
```

To create one, only the text needs to be sent. The timestamp is set automatically, the ID is created automatically, and the username is detected by the authentication data.

As this is an example, this API is designed to be minimal. In particular, more administrator endpoints could be created to effectively impersonate a user and allow administrator actions. The `DELETE` action was the first action included as a way of cleaning tests.

One final detail: there is some debate over whether it's best to end URI resources with a final slash or not. When working with Flask, though, defining them with a slash will return a redirect status code, `308 PERMANENT_REDIRECT`, for a request without the proper ending. In any case, try to be consistent to avoid confusion.

Defining the database schema

The database schema is simple and inherited from the monolith. We care only about the thoughts, stored in the `thought_model` table, so the database structure is as follows:

Field	Type	Comments
`id`	`INTEGER NOT NULL`	Primary key
`username`	`VARCHAR(50)`	
`text`	`VARCHAR(250)`	
`timestamp`	`DATETIME`	Creation time

The thought_model table

This table is represented in code in the `thoughts_backend/models.py` file, described in SQLAlchemy format with the following code:

```
class ThoughtModel(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50))
    text = db.Column(db.String(250))
    timestamp = db.Column(db.DateTime, server_default=func.now())
```

SQLAlchemy is capable of creating the table for testing purposes or for development mode. For this chapter, we defined the database to be SQLite, which stores the data in the `db.sqlite3` file.

Working with SQLAlchemy

SQLAlchemy (`https://www.sqlalchemy.org/`) is a powerful Python module to work with SQL databases. There are two approaches to dealing with databases with a high-level language such as Python. One is keeping the low-level approach and doing raw SQL statements, retrieving the data as it is in the database. The other is to abstract the database using an **Object-Relational Mapper** (**ORM**) and use the interface without getting into the details of how it is implemented.

The first approach is well represented by the Python database API specification (PEP 249—`https://www.python.org/dev/peps/pep-0249/`), which is followed by all major databases, such as `psycopg2` (`http://initd.org/psycopg/`) for PostgreSQL. This mainly creates SQL string commands, executes them, and then parses the results. This allows us to tailor each query, but it's not very productive for common operations that get repeated over and over. PonyORM (`https://ponyorm.org/`) is another example that's not so low level but still aims at replicating the SQL syntax and structure.

For the second approach, the best-known example is probably the Django ORM (`https://docs.djangoproject.com/en/2.2/topics/db/`). It abstracts the database access using defined model python objects. It works fantastically well for common operations, but its model assumes that the definition of the database is done in our Python code, and mapping legacy databases can be very painful. Some complex SQL operations created by the ORM can take a lot of time, while a custom-tailored query could save a lot of time. It's also easy to perform slow queries without even realizing, just because the tool abstracts us so much from the end result.

SQLAlchemy (`https://www.sqlalchemy.org/`) is quite flexible and can work on both ends of the spectrum. It's not as straightforward or as easy to use as the Django ORM, but it allows us to map existing databases into an ORM. This is why we will use it in our example: it can take an existing, complicated legacy database and map it, allowing you to perform simple operations easily and complicated operations in exactly the way you want.

Keep in mind that the operations we are going to be using in this book are quite simple and SQLAlchemy won't shine particularly in those tasks. But it's an invaluable tool if you're planning a complex migration from an old monolith that accesses the database through manually written SQL statements, to a newly created microservice. If you are already dealing with a complicated database, spending some time learning how to use SQLAlchemy will be invaluable. A well-tailored SQLAlchemy definition can perform some abstract tasks very efficiently, but it requires good knowledge of the tool.

The documentation for Flask-SQLAlchemy (`https://flask-sqlalchemy.palletsprojects.com/en/2.x/`) is a good place to start, as it summarizes the main operations, and the main SQLAlchemy documentation can be overwhelming at first.

After we define a model, we can perform a query by using the `query` attribute in the model and filter accordingly:

```
# Retrieve a single thought by its primary key
thought = ThoughtModel.query.get(thought_id)
# Retrieve all thoughts filtered by a username
thoughts = ThoughtModel.query.filter_by(username=username)
.order_by('id').all()
```

Storing and deleting a row requires the use of the session and then committing it:

```
# Create a new thought
new_thought = ThoughtModel(username=username, text=text,
timestamp=datetime.utcnow())
db.session.add(new_thought)
db.session.commit()

# Retrieve and delete a thought
thought = ThoughtModel.query.get(thought_id)
db.session.delete(thought)
db.session.commit()
```

To see how to configure the database access, check the `thoughts_backend/db.py` file.

Implementing the service

To implement this microservice, we will use Flask-RESTPlus (`https://flask-restplus.readthedocs.io/en/stable/`). This is a Flask (`https://palletsprojects.com/p/flask/`) extension. Flask is a well-known Python microframework for web applications that's particularly good at implementing microservices, as it's small, easy to use, and compatible with the usual technology stack in terms of web applications, since it uses the **Web Server Gateway Interface** (**WSGI**) protocol.

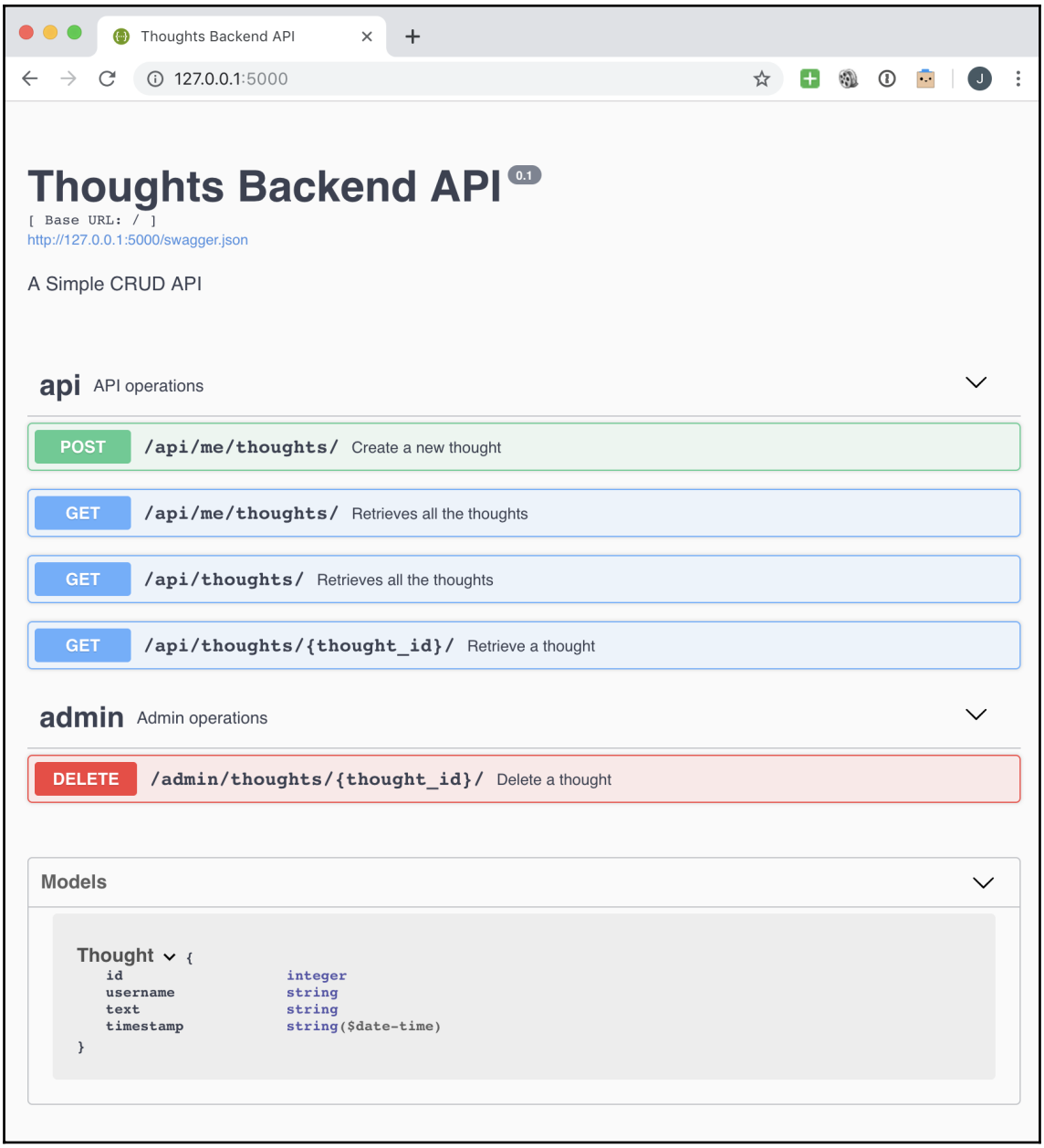
Introducing Flask-RESTPlus

Flask is capable of implementing a RESTful interface, but Flask-RESTPlus adds some very interesting capabilities that allow for good developing practices and speed of development:

- It defines namespaces, which are ways of creating prefixes and structuring the code. This helps long-term maintenance and helps with the design when creating new endpoints.

If you have more than 10 endpoints in a single namespace, it may be a good time to consider dividing it. Use one namespace per file, and allow the size of the file to hint when it's a good idea to try to make a division.

- It has a full solution for parsing input parameters. This means that we have an easy way of dealing with endpoints that requires several parameters and validates them. Using the *Request Parsing* (`https://flask-restplus.readthedocs.io/en/stable/parsing.html`) module is similar to using the `argparse` command-line module (`https://docs.python.org/3/library/argparse.html`) that's included in the Python standard library. It allows defining arguments in the body of the request, headers, query strings, or even cookies.
- In the same way, it has a serialization framework for the resulting objects. Flask-RESTful calls it **response marshalling** (`https://flask-restplus.readthedocs.io/en/stable/marshalling.html`). This helps to define objects that can be reused, clarifying the interface and simplifying the development. If enabled, it also allows for field masks, which return partial objects.
- It has full Swagger API documentation support. Swagger (`https://swagger.io/`) is an open source project to help in the design, implementation, documentation, and testing of RESTful API web services, following standard OpenAPI specifications. Flask-RESTPlus automatically generates a Swagger specification and self-documenting page:



The main Swagger documentation page for the Thoughts Backend API, generated automatically

Other nice elements of Flask are derived from the fact that it's a popular project and has a lot of supported tools:

- We will use the connector for SQLAlchemy, Flask-SQLAlchemy (`https://flask-sqlalchemy.palletsprojects.com/en/2.x/`). Its documentation covers most of the common cases, while the SQLAlchemy documentation is more detailed and can be a bit overwhelming.
- To run the tests, the `pytest-flask` module (`https://pytest-flask.readthedocs.io/en/latest/`) creates some fixtures ready to work with a Flask application. We will talk more about this in the *Testing the code* section.

Handling resources

A typical RESTful application has the following general structure:

1. A URL-defined **resource**. This resource allows one or more actions through HTTP methods (`GET`, `POST`, and so on).
2. When each of the actions is called, the framework routes the request until the defined code executes the action.
3. If there are any input parameters, they'll need to be validated first.
4. Perform the action and obtain a result value. This action will normally involve one or more calls to the database, which will be done in the shape of models.
5. Prepare the resulting result value and encode it in a way that's understood by the client, typically in JSON.
6. Return the encoded value to the client with the adequate status code.

Most of these actions are done by the framework. Some configuration work needs to be done, but it's where our web framework, Flask-RESTPlus in this example, will help the most. In particular, everything but *step 4* will be greatly simplified.

Let's take a look at a simple code example (available in GitHub) to describe it:

```
api_namespace = Namespace('api', description='API operations')

@api_namespace.route('/thoughts/<int:thought_id>/')
class ThoughtsRetrieve(Resource):

    @api_namespace.doc('retrieve_thought')
    @api_namespace.marshal_with(thought_model)
    def get(self, thought_id):
        '''
        Retrieve a thought
```

```
    '''
    thought = ThoughtModel.query.get(thought_id)
    if not thought:
        # The thought is not present
        return '', http.client.NOT_FOUND

    return thought
```

This implements the `GET /api/thoughts/X/` action, retrieving a single thought by ID.

Let's analyze each of the elements. Note the lines are grouped thematically:

1. First, we define the resource by its URL. Note that `api_namespace` sets the `api` prefix to the URL, which validates that parameter `X` is an integer:

```
api_namespace = Namespace('api', description='API operations')

@api_namespace.route('/thoughts/<int:thought_id>/')
class ThoughtsRetrieve(Resource):
    ...
```

2. The class allows you to perform multiple actions on the same resource. In this case, we only do one: the `GET` action.
3. Note that the `thought_id` parameter, encoded in the URL, is passed as a parameter to the method:

```
class ThoughtsRetrieve(Resource):

    def get(self, thought_id):
        ...
```

4. We can now execute the action, which is a search in the database to retrieve a single object. Call `ThoughModel` to search for the specified thought. If found, it's returned with a `http.client.OK (200)` status code. If it's not found, an empty result and a `http.client.NOT_FOUND 404` status code is returned:

```
def get(self, thought_id):
    thought = ThoughtModel.query.get(thought_id)
    if not thought:
        # The thought is not present
        return '', http.client.NOT_FOUND

    return thought
```

5. The `thought` object is being returned. The `marshal_with` decorator describes how the Python object should be serialized into a JSON structure. We'll see later how to configure it:

```
@api_namespace.marshal_with(thought_model)
def get(self, thought_id):
    ...
    return thought
```

6. Finally, we have some documentation, including the docstring that will be rendered by the autogenerated Swagger API:

```
class ThoughtsRetrieve(Resource):

    @api_namespace.doc('retrieve_thought')
    def get(self, thought_id):
        '''
        Retrieve a thought
        '''
        ...
```

As you can see, most of the actions are configured and performed through Flask-RESTPlus, and the bulk of the work as a developer is the meaty *step 4*. There's work to do, though, configuring what the expected input parameters are and validating them, as well as how to serialize the returning object into proper JSON. We'll see how Flask-RESTPlus can help us with that.

Parsing input parameters

The input parameters can take different shapes. When we talk about input parameters, we talk mainly about two kinds:

- String query parameters encoded into the URL. These are normally used for the `GET` requests, and look like the following:

```
http://test.com/some/path?param1=X&param2=Y
```

  They are part of the URL and will be stored in any log along the way. The parameters are encoded into their own format, called **URL encoding** (`https://www.urlencoder.io/learn/`). You've probably noticed that, for example, an empty space gets transformed to `%20`.

Normally, we won't have to decode query parameters manually, as frameworks such as Flask do it for us, but the Python standard library has utilities to do so (`https://docs.python.org/3/library/urllib.parse.html`).

- Let's look at the body of the HTTP request. This is typically used in the `POST` and `PUT` requests. The specific format can be specified using the `Content-Type` header. By default, the `Content-Type` header is defined as `application/x-www-form-urlencoded`, which encodes it in URL encoding. In modern applications, this is replaced with `application/json` to encode them in JSON.

The body of the requests is not stored in a log. The expectation is that a `GET` request produce the same result when called multiple times, that means they are idempotent. Therefore, it can be cached by some proxies or other elements. That's the reason why your browser asks for confirmation before sending a `POST` request again, as this operation may generate different results.

But there are two other places to pass parameters that can also be used:

- **As a part of the URL**: Things such as `thought id` are parameters. Try to follow RESTful principles and define your URLs as resources to avoid confusion. Query parameters are best left as optional.
- **Headers**: Normally, headers give information about metadata, such as the format of the request, the expected format, or authentication data. But they need to be treated as input parameters as well.

All of these elements are decoded automatically by Flask-RESTPlus, so we don't need to deal with encodings and low-level access.

Let's see how this works in our example. This code is extracted from the one in GitHub, and shortened to describe the parsing parameters:

```
authentication_parser = api_namespace.parser()
authentication_parser.add_argument('Authorization',
location='headers', type=str, help='Bearer Access
Token')

thought_parser = authentication_parser.copy()
thought_parser.add_argument('text', type=str, required=True, help='Text of
the thought')

@api_namespace.route('/me/thoughts/')
class MeThoughtListCreate(Resource):
```

```
    @api_namespace.expect(thought_parser)
    def post(self):
        args = thought_parser.parse_args()
        username = authentication_header_parser(args['Authorization'])
        text=args['text']
        ...
```

We define a parser in the following lines:

```
authentication_parser = api_namespace.parser()
authentication_parser.add_argument('Authorization',
location='headers', type=str, help='Bearer Access Token')

thought_parser = authentication_parser.copy()
thought_parser.add_argument('text', type=str, required=True, help='Text of
the thought')
```

`authentication_parser` is inherited by `thought_parser` to extend the functionality and combine both. Each of the parameters is defined in terms of type and whether they are required or not. If a required parameter is missing or another element is incorrect, Flask-RESTPlus will raise a `400 BAD_REQUEST` error, giving feedback about what went wrong.

Because we want to handle the authentication in a slightly different way, we label it as not required and allow it to use the default (as created for the framework) value of `None`. Note that we specify that the `Authorization` parameter should be in the headers.

The `post` method gets a decorator to show that it expects the `thought_parser` parameter, and we parse it with `parse_args`:

```
@api_namespace.route('/me/thoughts/')
class MeThoughtListCreate(Resource):

    @api_namespace.expect(thought_parser)
    def post(self):
        args = thought_parser.parse_args()
        ...
```

Furthermore, `args` is now a dictionary with all the parameters properly parsed and used in the next lines.

In the particular case of the authentication header, there's a specific function to work with that, and it return a `401 UNAUTHORIZED` status code through the usage of `abort`. This call immediately stops a request:

```
def authentication_header_parser(value):
    username = validate_token_header(value, config.PUBLIC_KEY)
    if username is None:
        abort(401)
    return username


class MeThoughtListCreate(Resource):

    @api_namespace.expect(thought_parser)
    def post(self):
       args = thought_parser.parse_args()
       username = authentication_header_parser(args['Authentication'])
       ...
```

We will leave aside for a moment the action to be performed (storing a new thought in the database), and focus on the other framework configuration, to serialize the result into a JSON object.

Serializing results

We need to return our results. The easiest way to do so is by defining the shape the JSON result should have through a serializer or marshalling model (`https://flask-restplus.readthedocs.io/en/stable/marshalling.html`).

A serializer model is defined as a dictionary with the expected fields and a field type:

```
from flask_restplus import fields

model = {
    'id': fields.Integer(),
    'username': fields.String(),
    'text': fields.String(),
    'timestamp': fields.DateTime(),
}
thought_model = api_namespace.model('Thought', model)
```

The model will take a Python object, and convert each of the attributes into the corresponding JSON element, as defined in the field:

```
@api_namespace.route('/me/thoughts/')
class MeThoughtListCreate(Resource):

    @api_namespace.marshal_with(thought_model)
    def post(self):
        ...
        new_thought = ThoughtModel(...)
        return new_thought
```

Note that `new_thought` is a `ThoughtModel` object, as retrieved by SQLAlchemy. We'll see it in detail next, but for now, it suffices to say that it has all the attributes defined in the model: `id`, `username`, `text`, and `timestamp`.

Any attribute not present in the memory object will have a value of `None` by default. You can change this default to a value that will be returned. You can specify a function, so it will be called to retrieve a value when the response is generated. This is a way of adding dynamic information to your object:

```
model = {
    'timestamp': fields.DateTime(default=datetime.utcnow),
}
```

You can also add the name of the attribute to be serialized, in case it's different than the expected outcome, or add a `lambda` function that will be called to retrieve the value:

```
model = {
    'thought_text': fields.String(attribute='text'),
    'thought_username': fields.String(attribute=lambda x: x.username),
 }
```

For more complex objects, you can nest values like this. Note this defines two models from the point of view of the documentation and that each `Nested` element creates a new scope. You can also use `List` to add multiple instances of the same kind:

```
extra = {
   'info': fields.String(),
}
extra_info = api_namespace.model('ExtraInfo', extra)

model = {
    'extra': fields.Nested(extra),
    'extra_list': fields.List(fields.Nested(extra)),
 }
```

Some of the available fields have more options, such as the date format for the `DateTime` fields. Check the full field's documentation (`https://flask-restplus.readthedocs.io/en/stable/api.html#models`) for more details.

If you return a list of elements, add the `as_list=True` parameter in the `marshal_with` decorator:

```
@api_namespace.route('/me/thoughts/')
class MeThoughtListCreate(Resource):

    @api_namespace.marshal_with(thought_model, as_list=True)
    def get(self):
        ...
        thoughts = (
            ThoughtModel.query.filter(
                ThoughtModel.username == username
            )
            .order_by('id').all()
        )
        return thoughts
```

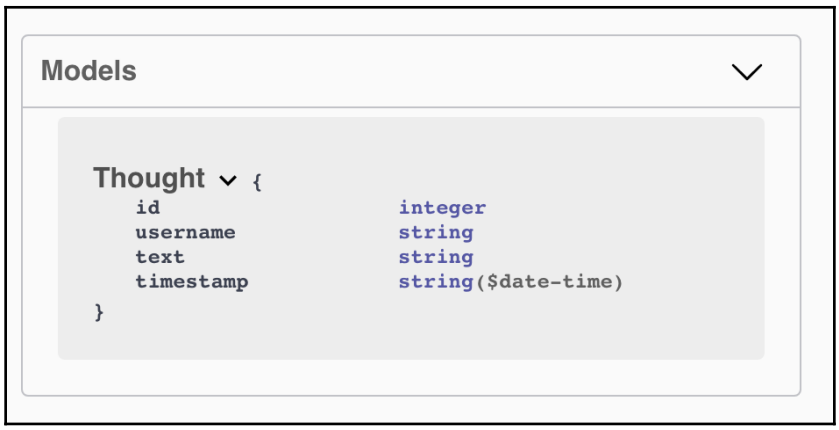
The `marshal_with` decorator will transform the `result` object from a Python object into the corresponding JSON data object.

By default, it will return a `http.client.OK (200)` status code, but we can return a different status code returning two values: the first is the object to `marshal` and the second is the status code. The code parameter in the `marshal_with` decorator is used for documentation purposes. Note, in this case, we need to add the specific `marshal` call:

```
@api_namespace.route('/me/thoughts/')
class MeThoughtListCreate(Resource):

    @api_namespace.marshal_with(thought_model,
         code=http.client.CREATED)
    def post(self):
        ...
        result = api_namespace.marshal(new_thought, thought_model)
        return result, http.client.CREATED
```

The Swagger documentation will display all your used-defined `marshal` objects:



The end of the Swagger page

One inconvenience of Flask-RESTPlus is that to input and output the same objects, they need to be defined twice, as the modules for input and output are different. This is not the case in some other RESTful frameworks, for example, in the Django REST framework (`https://www.django-rest-framework.org/`). The maintainers of Flask-RESTPlus are aware of this, and, according to them, they'll be integrating an external module, probably `marshmallow` (`https://marshmallow.readthedocs.io/en/stable/`). You can integrate it manually if you like, as Flask is flexible enough to do so, take a look at this example (`https://marshmallow.readthedocs.io/en/stable/examples.html#quotes-api-flask-sqlalchemy`).

For more details, you can check the full marshalling documentation at `https://flask-restplus.readthedocs.io/en/stable/marshalling.html`) of Flask-RESTPlus.

Performing the action

Finally, we get to the specific part where the input data is clean and ready to use, and we know how to return the result. This part likely involves performing some database query or queries and composing the results. Let's look at the following as an example:

```
@api_namespace.route('/thoughts/')
class ThoughtList(Resource):
```

```
@api_namespace.doc('list_thoughts')
@api_namespace.marshal_with(thought_model, as_list=True)
@api_namespace.expect(search_parser)
def get(self):
    '''
    Retrieves all the thoughts
    '''
    args = search_parser.parse_args()
    search_param = args['search']
    # Action
    query = ThoughtModel.query
    if search_param:
        query =(query.filter(
            ThoughtModel.text.contains(search_param)))

    query = query.order_by('id')
    thoughts = query.all()
    # Return the result
    return thoughts
```

You can see here, after parsing the parameters, we use SQLAlchemy to retrieve a query that, if the `search` parameter is present, will apply a filter. We obtain all the results with `all()`, returning all the `ThoughtModel` objects.

Returning the objects marshals (encodes them into JSON) them automatically, as we specified in the `marshal_with` decorator.

Authenticating the requests

The logic for authentication is encapsulated in the `thoughts_backend/token_validation.py` file. This contains both the generation and the validation of the header.

The following functions generate the `Bearer` token:

```
def encode_token(payload, private_key):
    return jwt.encode(payload, private_key, algorithm='RS256')

def generate_token_header(username, private_key):
    '''
    Generate a token header base on the username.
    Sign using the private key.
    '''
    payload = {
        'username': username,
```

```
            'iat': datetime.utcnow(),
            'exp': datetime.utcnow() + timedelta(days=2),
        }
        token = encode_token(payload, private_key)
        token = token.decode('utf8')
        return f'Bearer {token}'
```

This generates a JWT payload. It includes `username` to be used as a custom value, but it also adds two standard fields, an `exp` expiration date and the `iat` generation time of the token.

The token is then encoded using the RS256 algorithm, with a private key, and returned in the proper format: `Bearer <token>`.

The reverse action is to obtain the username from an encoded header. The code here is longer, as we should account for the different options in which we may receive the `Authentication` header. This header comes directly from our public API, so we should expect any value and program to be defensively ready for it.

The decoding of the token itself is straightforward, as the `jwt.decode` action will do this:

```
def decode_token(token, public_key):
    return jwt.decode(token, public_key, algoritms='RS256')
```

But before arriving at that step, we need to obtain the token and verify that the header is valid in multiple cases, so we check first whether the header is empty, and whether it has the proper format, extracting the token:

```
def validate_token_header(header, public_key):
    if not header:
        logger.info('No header')
        return None

    # Retrieve the Bearer token
    parse_result = parse('Bearer {}', header)
    if not parse_result:
        logger.info(f'Wrong format for header "{header}"')
        return None
    token = parse_result[0]
```

Then, we decode the token. If the token cannot be decoded with the public key, it raises `DecodeError`. The token can also be expired:

```
        try:
            decoded_token = decode_token(token.encode('utf8'), public_key)
        except jwt.exceptions.DecodeError:
            logger.warning(f'Error decoding header "{header}". '
```

```
            'This may be key missmatch or wrong key')
            return None
    except jwt.exceptions.ExpiredSignatureError:
            logger.info(f'Authentication header has expired')
            return None
```

Then, check that it has the expected `exp` and `username` parameters. If any of these parameters is missing, that means that the token format, after decoding, is incorrect. This may happen when changing the code in different versions:

```
    # Check expiry is in the token
    if 'exp' not in decoded_token:
        logger.warning('Token does not have expiry (exp)')
        return None

    # Check username is in the token
    if 'username' not in decoded_token:
        logger.warning('Token does not have username')
        return None

    logger.info('Header successfully validated')
    return decoded_token['username']
```

If everything goes fine, return the username at the end.

Each of the possible problems is logged with a different severity. Most common occurrences are logged with info- level security, as they are not grave. Things such as a format error after the token is decoded may indicate a problem with our encoding process.

Note that we are using a private/public key schema, instead of a symmetric key schema, to encode and decode the tokens. This means that the decoding and encoding keys are different.

Technically, this is a sign/verification as it is used to generate a signature, and not encode/decode, but it's the naming convention used in JWT.

In our microservice structure, only the signing authority requires the private key. This increases the security as any key leakage in other services won't be able to retrieve a key capable of signing bearer tokens. We'll need to generate proper private and public keys, though.

To generate a private/public key, run the following command:

```
$ openssl genrsa -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
.....................+++
.............................+++
```

Then, to extract the public key, use the following:

```
$ openssl rsa -in key.pem -outform PEM -pubout -out key.pub
```

This will generate two files: `key.pem` and `key.pub` with a private/public key pair. Reading them in text format will be enough to use them as keys for encoding/decoding the JWT token:

```
>> with open('private.pem') as fp:
>> ..  private_key = fp.read()

>> generate_token_header('peter', private_key)
'Bearer <token>'
```

Note that, for the tests, we generated a **sample key pair** that's attached as strings. These keys have been created specifically for this usage and are not used anywhere else. Please do not use them anywhere as they are publicly available in GitHub.

Be aware that you require a non-encrypted private key, not protected by a password, as the JWT module doesn't allow you to add a password. **Do not store production secret keys in unprotected files**. In `Chapter 3`, *Build, Run, and Test Your Service Using Docker*, we'll see how to inject this secret using an environment variable, and in `Chapter 11`, *Handling Change, Dependencies, and Secrets in the System*, we'll see how to properly deal with secrets in production environments.

Testing the code

To test our application, we use the excellent `pytest` framework, which is the gold standard in test runners for Python applications.

Basically, `pytest` has a lot of plugins and add-ons to deal with a lot of situations. We will be using `pytest-flask`, which helps with running tests for Flask applications.

To run all the tests, just call `pytest` in the command line:

```
$ pytest
============== test session starts ==============
....
==== 17 passed, 177 warnings in 1.50 seconds =====
```

Note that `pytest` has a lot of features available to deal with a lot of situations while testing. Things running a subset of matched tests (the `-k` option), running the last failed tests (`--lf`), or stopping after the first failure (`-x`) are incredibly useful when working with tests. I highly recommend checking its full documentation (`https://docs.pytest.org/en/latest/`) and discovering all its possibilities.

There are also a lot of plugins and extensions for using databases or frameworks, reporting code coverage, profiling, BDD, and many others. It is worth finding out about them.

We configure the basic usage, including always enabling flags in the `pytest.ini` file and the fixtures in `conftest.py`.

Defining the pytest fixtures

Fixtures are used in `pytest` to prepare the context in which a test should be executed, preparing it and cleaning it at the end. The application fixture is expected by `pytest-flask`, as seen in the documentation. The plugin generates a `client` fixture that we can use to send requests in test mode. We see this fixture in action in the `thoughts_fixture` fixture, which generates three thoughts through the API and deletes everything after our test has run.

The structure, simplified, is as follows:

1. Generate three thoughts. Store its `thought_id`:

```
@pytest.fixture
def thought_fixture(client):

    thought_ids = []
    for _ in range(3):
        thought = {
            'text': fake.text(240),
        }
        header =
```

```
token_validation.generate_token_header(fake.name(),
PRIVATE_KEY)
        headers = {
            'Authorization': header,
        }
        response = client.post('/api/me/thoughts/', data=thought,
                               headers=headers)
        assert http.client.CREATED == response.status_code
        result = response.json
        thought_ids.append(result['id'])
```

2. Then, add `yield thought_ids` to the test:

```
yield thought_ids
```

3. Retrieve all thoughts and delete them one by one:

```
# Clean up all thoughts
response = client.get('/api/thoughts/')
thoughts = response.json
for thought in thoughts:
    thought_id = thought['id']
    url = f'/admin/thoughts/{thought_id}/'
    response = client.delete(url)
    assert http.client.NO_CONTENT == response.status_code
```

Note that we use the `faker` module to generate fake names and text. You can check its full documentation at `https://faker.readthedocs.io/en/stable/`. It is a great way of generating random values for your tests that avoid reusing `test_user` and `test_text` over and over. It also helps to shape your tests, by checking the input independently and not blindly copying a placeholder.

Fixtures can also exercise your API. You can choose a lower-level approach such as writing raw information in your database, but using your own defined API is a great way of ensuring that you have a complete and useful interface. In our example, we added an admin interface that's used to delete thoughts. This is exercised throughout the fixture as well as the creation of thoughts for a whole and complete interface.

This way, we also use tests to validate that we can use our microservice as a complete service, without tricking ourselves into hacking our way to perform common operations.

Also note the usage of the `client` fixture, which is provided by `pytest-flask`.

Understanding test_token_validation.py

This test file tests the behavior of the `token_validation` module. This module covers the generation and validation of the authentication header, so it's important to test it thoroughly.

The tests check that the header can be encoded and decoded with the proper keys. It also checks all the different possibilities in terms of invalid inputs: different shapes of incorrect formats, invalid decoding keys, or expired tokens.

To check for expired tokens, we use two modules: `freezegun`, to make the test to retrieve a specific test time (`https://github.com/spulec/freezegun`), and `delorean`, to parse dates easily (though, the module is capable of way more; check the documentation at `https://delorean.readthedocs.io/en/latest/`). These two modules are very easy to use and great for testing purposes.

For example, this test checks an expired token:

```
@freeze_time('2018-05-17 13:47:34')
def test_invalid_token_header_expired():
    expiry = delorean.parse('2018-05-17 13:47:33').datetime
    payload = {
        'username': 'tonystark',
        'exp': expiry,
    }
    token = token_validation.encode_token(payload, PRIVATE_KEY)
    token = token.decode('utf8')
    header = f'Bearer {token}'
    result = token_validation.validate_token_header(header, PUBLIC_KEY)
    assert None is result
```

Note how the freeze time is precisely 1 second after the expiry time of the token.

The public and private keys used for tests are defined in the `constants.py` file. There's an extra independent public key used to check what happens if you decode a token with an invalid public key.

It is worth saying it again: please *do not* use any of these keys. These keys are for running tests only and are available to anyone who has access to this book.

test_thoughts.py

This file checks the defined API interfaces. Each API is tested to perform the actions correctly (create a new thought, return thoughts of a user, retrieve all thoughts, search through thoughts, and retrieve a thought by ID) as well as some error tests (unauthorized requests to create and retrieve thoughts of a user, or retrieve a non-existing thought).

Here, we use `freezegun` again to determine when the thoughts are created, instead of creating them with a timestamp dependent on the time when tests are run.

Summary

In this chapter, we saw how to develop a web microservice. We started by designing its API following REST principles. Then, we described how to access the schema of the database, and how to do it using SQLAlchemy.

Then, we learned how to implement it using Flask-RESTPlus. We learned how to define the resources being mapped to the API endpoints, how to parse the input values, how to process the actions, and then how to return the results using the serializer model. We described how the authentication layer works.

We included tests and described how to use the `pytest` fixture to create initial conditions for our tests. In the next chapter, we will look at how to containerize the service and run it through Docker.

Questions

1. Can you name the characteristics of RESTful applications?
2. What are the advantages of using Flask-RESTPlus?
3. Which alternative frameworks to Flask-RESTPlus do you know?
4. Name the Python package used in the tests to fix the time.
5. Can you describe the authentication flow?
6. Why did we choose SQLAlchemy as a database interface for the example project?

Further reading

For an in-depth description of a RESTful design that is not limited to Python, you can find more information in *Hands-On RESTful API Design Patterns and Best Practices* (`https://www.packtpub.com/gb/application-development/hands-restful-api-design-patterns-and-best-practices`). You can learn more about how to use the Flask framework in the book *Flask: Building Python Web Services* (`https://www.packtpub.com/gb/web-development/flask-building-python-web-services`).

3
Build, Run, and Test Your Service Using Docker

Having designed a working RESTful microservice in the previous chapter, we'll see in this chapter how to use it in *the Docker way*, encapsulating the service into a self-contained container so that it's immutable and can be deployed on its own. This chapter describes very explicitly the dependencies of the service and the ways it can be used. The main way to run a service is to run it as a web server, but other operations are possible, such as running unit tests, generating reports, and others. We'll see also how to deploy the service on your local computer for testing and how to share it through an image repository.

The following topics will be covered in this chapter:

- Building your service with a Dockerfile
- Operating with an immutable container
- Configuring your service
- Deploying the Docker service locally
- Pushing your Docker image to a remote registry

By the end of the chapter, you'll know how to operate with Docker, create a basic service, build an image, and run it. You'll also know how to share the image to be run on another computer.

Technical requirements

For this chapter, you need to install Docker, version 18.09 or above. See the official documentation (`https://docs.docker.com/install/`) for how to do so for your platform.

If you install Docker in Linux, you may have to configure the server to run for non-root access. Check the documentation at `https://docs.docker.com/install/linux/linux-postinstall/`.

Check the version with the following command:

```
$ docker version
Client: Docker Engine - Community
 Version: 18.09.2
 API version: 1.39
 Go version: go1.10.8
 Git commit: 6247962
 Built: Sun Feb 10 04:12:39 2019
 OS/Arch: darwin/amd64
 Experimental: false
```

You need to install Docker Compose version 1.24.0 or above as well. Note that, in some installations, such as macOS, this is automatically installed for you. Check the installation instructions in the Docker documentation (`https://docs.docker.com/compose/install/`):

```
$ docker-compose version
docker-compose version 1.24.0, build 0aa5906
docker-py version: 3.7.2
CPython version: 3.7.3
OpenSSL version: OpenSSL 1.0.2r 26 Feb 2019
```

The code is available on GitHub, in this directory: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter03`. There's a copy of `ThoughtsBackend` presented in `Chapter 2`, *Creating a REST Service with Python*, but the code is slightly different. We will look at the differences in this chapter.

Building your service with a Dockerfile

It all starts with a container. As we said in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, containers are a packetized bundle of software, encapsulated in a standard way. They are units of software that can be run independently, as they are totally self-contained. To make a container, we need to build it.

Remember our description of a container as a process surrounded by its own filesystem. Building a container constructs this filesystem.

To build a container with Docker, we need a definition of its content. The filesystem is created by applying layer after layer. Each Dockerfile, the recipe for generating a container, contains a definition of steps to generate a container.

For example, let's create a very simple Dockerfile. Create a file called `example.txt` with some example text and another called `Dockerfile.simple` with the following:

```
# scratch is a special container that is totally empty
FROM scratch
COPY example.txt /example.txt
```

Now build it using the following command:

```
$ # docker build -f <dockerfile> --tag <tag> <context>
$   docker build -f Dockerfile.simple --tag simple .
Sending build context to Docker daemon 3.072kB
Step 1/2 : FROM scratch
 --->
Step 2/2 : COPY example.txt /example.txt
 ---> Using cache
 ---> f961aef9f15c
Successfully built f961aef9f15c
Successfully tagged simple:latest

$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
simple latest f961aef9f15c 4 minutes ago 11B
```

This creates a Docker image that only contains the `example.txt` file. It's not very useful, but quite small—only 11 bytes. That's because it inherits from the empty container, `scratch`. It then copies the `example.txt` file inside the location in the `/example.txt` container.

Let's take a look at the `docker build` command. The Dockerfile is defined with the `-f` parameter, the tag of the resulting image is defined with `--tag` , and the `context` parameter is defined as dot (`.`). The `context` parameter is the reference to where to look for the files defined in the steps in the Dockerfile.

The image also has the image ID `f961aef9f15c`, which is assigned automatically. This is a hash of the contents of the filesystem. We'll see later why this is relevant.

Executing commands

The previous container was not very exciting. It is definitively possible to create your own container totally from scratch, but, typically, you'll look for a baseline that contains some sort of Linux distribution that allows you to do something useful with the container.

As we saw with the `FROM` command, we can start with a previous container. We will use the Alpine Linux (`https://alpinelinux.org/`) distribution throughout the book, though other distributions are available, such as Ubuntu and CentOS. Check out the article at `https://sweetcode.io/linux-distributions-optimized-hosting-docker/` for distributions aimed at Docker containers.

Why Alpine Linux? It is arguably the most popular distribution for Docker systems because it has a very small footprint and it's aimed at security. It is well-maintained and regularly updated and patched. It also has a complete package management system that allows you to install most of the common tools for web services easily. The base image is only around 5 MB in size and contains a working Linux operating system.

It has a couple of quirks when working with it, such as using its own package management, called `apk`, but it's easy to use and is almost a straight-on drop replacement for common Linux distributions.

The following Dockerfile will inherit from the base `alpine` container and add the `example.txt` file:

```
FROM alpine

RUN mkdir -p /opt/
COPY example.txt /opt/example.txt
```

This container allows us to run commands, as the usual command-line utilities are included:

```
$ docker build -f Dockerfile.run --tag container-run .
Sending build context to Docker daemon 4.096kB
Step 1/3 : FROM alpine
 ---> 055936d39205
Step 2/3 : RUN mkdir -p /opt/
 ---> Using cache
 ---> 4f565debb941
Step 3/3 : COPY example.txt /opt/example.txt
 ---> Using cache
 ---> d67a72454d75
Successfully built d67a72454d75
```

```
Successfully tagged container-run:latest

$ # docker run <image name> <command>
$   docker run container-run cat /opt/example.txt
An example file
```

Note how the `cat /opt/example.txt` command line gets executed. This is actually happening inside the container. We print the result in `stdout` in our `stdout` console. However, if there's a file created, as the container stops, the file is not saved in our local filesystem, but only inside container:

```
$ ls
Dockerfile.run example.txt
$ docker run container-run /bin/sh -c 'cat /opt/example.txt > out.txt'
$ ls
Dockerfile.run example.txt
```

The file is actually saved in a stopped container. Once the container has finished its run, it remains stopped by Docker until removed. You can see the stopped container with the `docker ps -a` command. A stopped container is not very interesting, though its filesystem is saved on disk.

When running web services, the command being run won't stop; it will keep running until stopped. Remember what we said before about a container being a process with a filesystem attached. The command running is the key to the container.

You can add a default command, which will be executed when no command is given, by adding the following:

```
CMD cat /opt/example.txt
```

Make it run automatically by using the following command:

```
$ docker run container-run
An example file
```

Defining a standard command makes the container really simple. Just run it and it will do whatever it is configured to do. Remember to include a default command in your containers.

We can also execute a shell in the container and interact with it. Remember to add the `-it` flag to keep the connection properly open, `-i` to keep `stdin` open, and `-t` to create a pseudo Terminal, you can remember it as interactive Terminal:

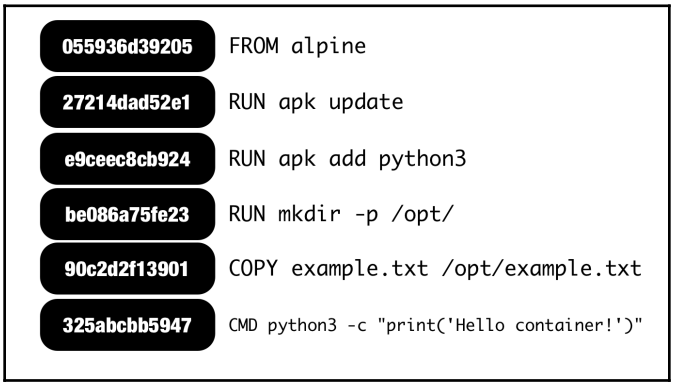
```
$ docker run -it container-run /bin/sh
/ # cd opt/
/opt # ls
example.txt
/opt # cat example.txt
An example file
/opt # exit
$
```

This is very useful when finding out problems or performing exploratory tests.

Understanding the Docker cache

One of the main points of confusion when building images is understanding how the Docker layers work.

Each of the commands on a Dockerfile is executed consecutively and on top of the previous layer. If you are comfortable with Git, you'll notice that the process is similar. Each layer only stores the changes to the previous step:



This allows Docker to cache quite aggressively, as any layer before a change is already calculated. For example, in this example, we update the available packages with `apk update`, then install the `python3` package, before copying the `example.txt` file. Any changes to the `example.txt` file will only execute the last two steps over layer `be086a75fe23`. This speeds up the rebuilding of images.

It also means that you need to construct your Dockerfiles carefully to not invalidate the cache. Start with the operations that change very rarely, such as installing the project dependencies, and finish with the ones that change more often, such as adding your code. The annotated Dockerfile for our example has indications about the usage of the cache.

This also means that an image will never get smaller in size, adding a new layer even if the layer removes data, as the previous layer is still stored on the disk. If you want to remove cruft from a step, you'll need to do so in the same step.

Keeping your containers small is quite important. In any Docker system, the tendency is to have a bunch of containers and lots of images. Big images for no reason will fill up repositories quickly. They'll be slow to download and push, and also slow to start, as the container is copied around in your infrastructure.

There's another practical consideration. Containers are a great tool to simplify and reduce your service to the minimum. With a bit of investment, you'll have great results and keep small and to-the-point containers.

There are several practices for keeping your images small. Other than being careful to not install extra elements, the main ones are creating a single, complicated layer that installs and uninstalls, and multi-stage images. Multi-stage Dockerfiles are a way of referring to a previous intermediate layer and copying data from there. Check the Docker documentation (`https://docs.docker.com/develop/develop-images/multistage-build/`).

Compilers, in particular, tend to get a lot of space. When possible, try to use precompiled binaries. You can use a multi-stage Dockerfile to compile in one container and then copy the binaries to the running one.

You can learn more about the differences between the two strategies in this article: `https://pythonspeed.com/articles/smaller-python-docker-images/`.

A good tool to analyze a particular image and the layers that compose it is `dive` (`https://github.com/wagoodman/dive`). It will also discover ways that an image can be reduced in size.

We'll create a multi-stage container in the next step.

Building a web service container

We have a specific objective, to create a container that is capable of running our microservice, `ThoughtsBackend`. To do so, we have a couple of requirements:

- We need to copy our code to the container.
- The code needs to be served through a web server.

So, in broad strokes, we need to create a container with a web server, add our code, configure it so it runs our code, and serve the result when starting the container.

We will store most of the configuration files inside subdirectories in the `./docker` directory.

As a web server, we will use uWSGI (`https://uwsgi-docs.readthedocs.io/en/latest/`). uWSGI is a web server capable of serving our Flask application through the WSGI protocol. uWSGI is quite configurable, has a lot of options, and is capable of serving HTTP directly.

A very common configuration is to have NGINX in front of uWSGI to serve static files, as it's more efficient for that. In our specific use case, we don't serve many static files, as we're running a RESTful API, and, in our main architecture, as described in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, there's already a load balancer on the frontend and a dedicated static files server. This means we won't be adding an extra component for simplicity. NGINX usually communicates to uWSGI using the `uwsgi` protocol, which is a protocol specifically for the uWSGI server, but it can also do it through HTTP. Check the NGINX and uWSGI documentation.

Let's take a look at the `docker/app/Dockerfile` file. It has two stages; the first one is to compile the dependencies:

```
########
# This image will compile the dependencies
# It will install compilers and other packages, that won't be carried
# over to the runtime image
########
FROM alpine:3.9 AS compile-image

# Add requirements for python and pip
RUN apk add --update python3
```

```
RUN mkdir -p /opt/code
WORKDIR /opt/code

# Install dependencies
RUN apk add python3-dev build-base gcc linux-headers postgresql-dev libffi-
dev

# Create a virtual environment for all the Python dependencies
RUN python3 -m venv /opt/venv
# Make sure we use the virtualenv:
ENV PATH="/opt/venv/bin:$PATH"
RUN pip3 install --upgrade pip

# Install and compile uwsgi
RUN pip3 install uwsgi==2.0.18
# Install other dependencies
COPY ThoughtsBackend/requirements.txt /opt/
RUN pip3 install -r /opt/requirements.txt
```

This stage does the following steps:

1. Names the stage `compile-image`, inheriting from Alpine.
2. Installs `python3`.
3. Installs the build dependencies, including the `gcc` compiler and Python headers (`python3-dev`).
4. Creates a new virtual environment. We will install all the Python dependencies here.
5. The virtual environment gets activated.
6. Installs uWSGI. This step compiles it from code.

You can also install the included uWSGI package in the Alpine distribution, but I found the compiled package to be more complete and easier to configure, as the Alpine `uwsgi` package requires you to install other packages such as `uwsgi-python3`, `uwsgi-http`, and so on, then enable the plugin in the uWSGI config. The size difference is minimal. This also allows you to use the latest uWSGI version and not depend on the one in your Alpine distribution.

7. Copy the `requirements.txt` file and install all the dependencies. This will compile and copy the dependencies to the virtual environment.

The second stage is preparing the running container. Let's take a look:

```
########
# This image is the runtime, will copy the dependencies from the other
########
FROM alpine:3.9 AS runtime-image

# Install python
RUN apk add --update python3 curl libffi postgresql-libs

# Copy uWSGI configuration
RUN mkdir -p /opt/uwsgi
ADD docker/app/uwsgi.ini /opt/uwsgi/
ADD docker/app/start_server.sh /opt/uwsgi/

# Create a user to run the service
RUN addgroup -S uwsgi
RUN adduser -H -D -S uwsgi
USER uwsgi

# Copy the venv with compile dependencies from the compile-image
COPY --chown=uwsgi:uwsgi --from=compile-image /opt/venv /opt/venv
# Be sure to activate the venv
ENV PATH="/opt/venv/bin:$PATH"

# Copy the code
COPY --chown=uwsgi:uwsgi ThoughtsBackend/ /opt/code/

# Run parameters
WORKDIR /opt/code
EXPOSE 8000
CMD ["/bin/sh", "/opt/uwsgi/start_server.sh"]
```

It carries out the following actions:

1. Labels the image as `runtime-image` and inherits from Alpine, as previously.
2. Installs Python and other requirements for the runtime.

Note that any runtime required for compilation needs to be installed. For example, we install `libffi` in the runtime and `libffi-dev` to compile, required by the `cryptography` package. A mismatch will raise a runtime error when trying to access the (non-present) libraries. The `dev` libraries normally contain the runtime libraries.

3. Copy the uWSGI configuration and script to start the service. We'll take a look at that in a moment.

4. Create a user to run the service, and set it as the default using the `USER` command.

This step is not strictly necessary as, by default, the root user will be used. As our containers are isolated, gaining root access in one is inherently more secure than in a real server. In any case, it's good practice to not configure our public-facing service accessing as root and it will remove some understandable warnings.

5. Copy the virtual environment from the `compile-image` image. This installs all the compiled Python packages. Note that they are copied with the user to run the service, to have access to them. The virtual environment is activated.
6. Copy the application code.
7. Define the run parameters. Note that port `8000` is exposed. This will be the port we will serve the application on.

If running as root, port `80` can be defined. Routing a port in Docker is trivial, though, and other than the front-facing load balancer, there's not really any reason why you need to use the default HTTP port. Use the same one in all your systems, though, which will remove uncertainty.

Note that the application code is copied at the end of the file. The application code is likely going to be the code that changes most often, so this structure takes advantage of the Docker cache and recreates only the very few last layers, instead of having to start from the beginning. Take this into account when designing your Dockerfiles.

Also, keep in mind that there's nothing stopping you from changing the order while developing. If you're trying to find a problem with a dependency, and so on, you can comment out irrelevant layers or add steps later once the code is stable.

Let's build our container now. See that there are two images created, though only one is named. The other is the compile image, which is much bigger as it contains the compilers, and so on:

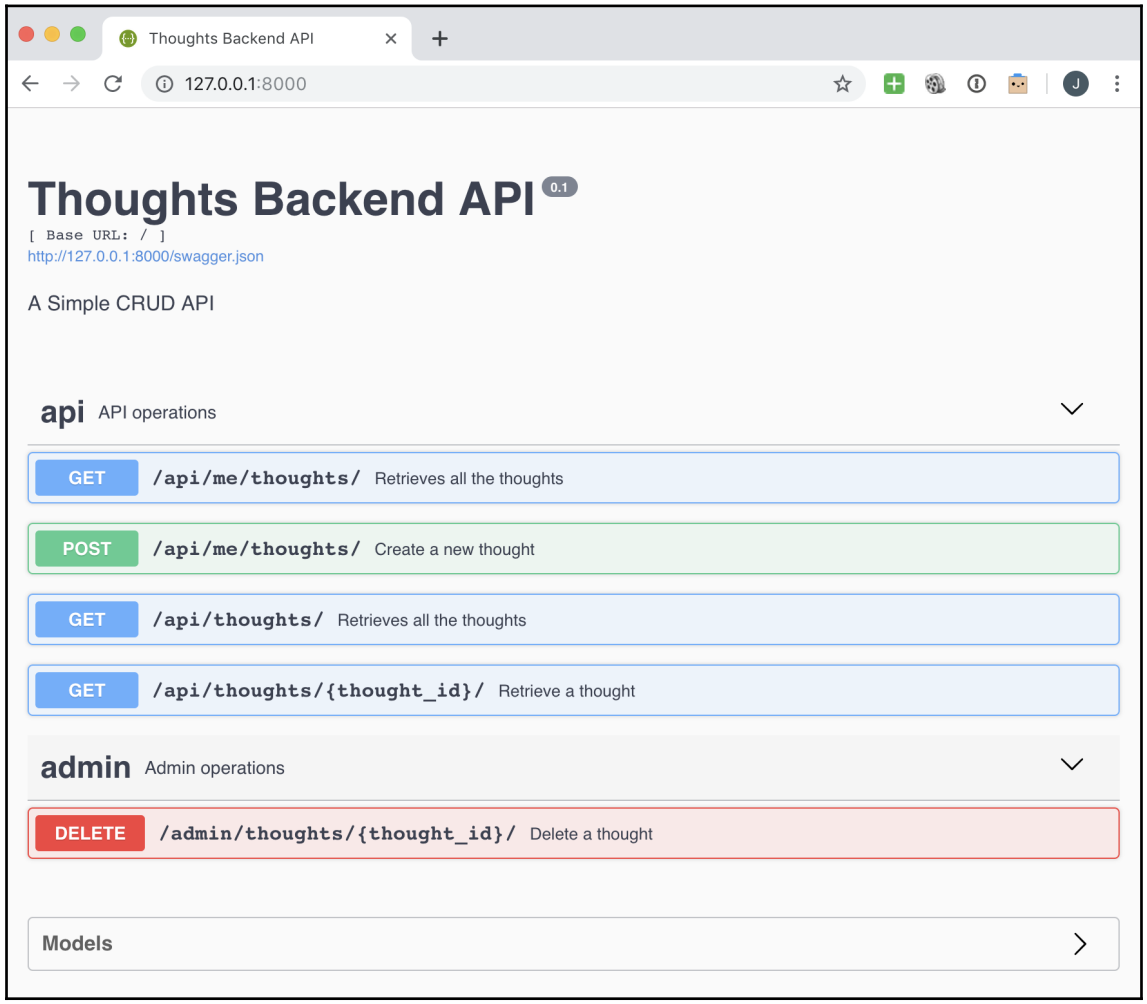
```
$ docker build -f docker/app/Dockerfile --tag thoughts-backend .
...
 ---> 027569681620
Step 12/26 : FROM alpine:3.9 AS runtime-image
...
Successfully built 50efd3830a90
Successfully tagged thoughts-backend:latest
$ docker images | head
```

```
REPOSITORY TAG IMAGE ID CREATED SIZE
thoughts-backend latest 50efd3830a90 10 minutes ago 144MB
<none>           <none> 027569681620 12 minutes ago 409MB
```

Now we can run the container. To be able to access the internal port `8000`, we need to route it with the `-p` option:

```
$ docker run -it  -p 127.0.0.1:8000:8000/tcp thoughts-backend
```

Accessing our local browser to `127.0.0.1` shows our application. You can see the access logs in the standard output:



You can access a running container from a different Terminal with `docker exec` and execute a new shell. Remember to add `-it` to keep the Terminal open. Inspect the currently running containers with `docker ps` to find the container ID:

```
$ docker ps
CONTAINER ID IMAGE            COMMAND ... PORTS ...
ac2659958a68 thoughts-backend ... ...     127.0.0.1:8000->8000/tcp
$ docker exec -it ac2659958a68 /bin/sh
/opt/code $ ls
README.md __pycache__ db.sqlite3 init_db.py pytest.ini requirements.txt
tests thoughts_backend wsgi.py
/opt/code $ exit
$
```

You can stop the container with *Ctrl + C*, or, more gracefully, stop it from another Terminal:

```
$ docker ps
CONTAINER ID IMAGE            COMMAND ... PORTS ...
ac2659958a68 thoughts-backend ... ...     127.0.0.1:8000->8000/tcp
$ docker stop ac2659958a68
ac2659958a68
```

The logs will show `graceful stop`:

```
...
spawned uWSGI master process (pid: 6)
spawned uWSGI worker 1 (pid: 7, cores: 1)
spawned uWSGI http 1 (pid: 8)
Caught SIGTERM signal! Sending graceful stop to uWSGI through the master-
fifo
Fri May 31 10:29:47 2019 - graceful shutdown triggered...
$
```

Capturing `SIGTERM` properly and stopping our services gracefully is important for avoiding abrupt terminations of services. We'll see how to configure this in uWSGI, as well as the rest of the elements.

Configuring uWSGI

The `uwsgi.ini` file contains the uWSGI configuration:

```
[uwsgi]
uid=uwsgi
chdir=/opt/code
wsgi-file=wsgi.py
master=True
```

```
pidfile=/tmp/uwsgi.pid
http=:8000
vacuum=True
processes=1
max-requests=5000
# Used to send commands to uWSGI
master-fifo=/tmp/uwsgi-fifo
```

Most of it is information that we have from the Dockerfile, though it needs to match so that uWSGI knows where to find the application code, the name of the WSGI file to start, the user to start it from, and so on.

Other parameters are specific to uWSGI behavior:

- `master`: Creates a master process that controls the others. Recommended for uWSGI operation as it creates smoother operation.
- `http`: Serves in the specified port. The HTTP mode creates a process that load balances the HTTP requests toward the workers, and it's recommended to serve HTTP outside of the container.
- `processes`: The number of application workers. Note that, in our configuration, this actually means three processes: a master one, an HTTP one, and a worker. More workers can handle more requests but will use more memory. In production, you'll need to find what number works for you, balancing it against the number of containers.
- `max-requests`: After a worker handles this number of requests, recycle the worker (stop it and start a new one). This reduces the probability of memory leaks.
- `vacuum`: Clean the environment when exiting.
- `master-fifo`: Create a Unix pipe to send commands to uWSGI. We will use this to handle graceful stops.

The uWSGI documentation (`https://uwsgi-docs.readthedocs.io/en/latest/`) is quite extensive and comprehensive. It contains a lot of valuable information, both for operating uWSGI itself and understanding details about how web servers operate. I learn something new each time that I read it, but it can be a bit overwhelming at first.

It's worth investing a bit of time in running tests to discover what are the best parameters for your service in areas such as timeouts, the number of workers, and so on. However, remember that some of the options for uWSGI may be better served with your container's configuration, which simplifies things.

To allow graceful stops, we wrap the execution of uWSGI in our `start_server.sh` script:

```
#!/bin/sh

_term() {
  echo "Caught SIGTERM signal! Sending graceful stop to uWSGI through the
master-fifo"
  # See details in the uwsgi.ini file and
  # in http://uwsgi-docs.readthedocs.io/en/latest/MasterFIFO.html
  # q means "graceful stop"
  echo q > /tmp/uwsgi-fifo
}

trap _term SIGTERM

uwsgi --ini /opt/uwsgi/uwsgi.ini &

# We need to wait to properly catch the signal, that's why uWSGI is started
# in the background. $! is the PID of uWSGI
wait $!
# The container exits with code 143, which means "exited because SIGTERM"
# 128 + 15 (SIGTERM)
# http://www.tldp.org/LDP/abs/html/exitcodes.html
# http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_12_02.html
```

The core of the script is the call to `uwsgi` to start the service. It will then wait until it stops.

The `SIGTERM` signal will be captured and uWSGI will be stopped gracefully by sending the `q` command to the `master-fifo` pipe.

A graceful stop means that a request won't be interrupted when a new container version is available. We'll see later how to make rollout deployments, but one of the key elements is to interrupt existing servers when they are not serving requests, to avoid stopping in the middle of a request and leaving an inconsistent state.

Docker uses the `SIGTERM` signal to stop the execution of containers. After a timeout, it will kill them with `SIGKILL`.

Refreshing Docker commands

We've looked at some of the important Docker commands:

- `docker build`: Builds an image
- `docker run`: Runs an image
- `docker exec`: Executes a command in a running container
- `docker ps`: Shows the currently running containers
- `docker images`: Displays the existing images

While these are the basic ones, knowing most of the available Docker commands is very useful for debugging problems and to perform operations such as monitoring, copying and tagging images, creating networks, and so on. These commands will also show you a lot about how Docker works internally.

An important command: be sure to clean up old containers and images with `docker system prune` from time to time. Docker is quite space-intensive after working with it for a few weeks.

The Docker documentation (`https://docs.docker.com/v17.12/engine/reference/commandline/docker/`) is quite complete. Be sure to know your way around it.

Operating with an immutable container

Docker commands such as the ones seen earlier in this chapter are the foundation, where it all starts. But, when dealing with more than one, it starts getting complicated to handle them. You've seen that some commands can get quite long.

To operate with a container in a clustered operation, we will use `docker-compose`. This is Docker's own orchestration tool for defining multi-container operations. It gets defined by a YAML file with all the different tasks and services, each with enough context to build and run it.

It allows you to store the different services and parameters for each of them in this configuration file, called `docker-compose.yaml` by default. This allows you to coordinate them and generate a replicable cluster of services.

Testing the container

We will start by creating a service to run the unit tests. Keep in mind that the tests need to run *inside* the container. This will standardize the execution of them and ensure that the dependencies are constant.

Note that, in the creation of our container, we include all the requirements to execute the tests. There's the option to create the running container and inherit from it to add the tests and test dependencies.

This certainly creates a smaller running container but creates a situation where the testing container is not 100% exactly the same as the one in production. If the size is critical and there's a big difference, this may be an option, but be aware of the differentiation if there's a subtle bug.

We need to define a service in the `docker-compose.yaml` file, in this way:

```
version: '3.7'

services:
    # Development related
    test-sqlite:
        environment:
            - PYTHONDONTWRITEBYTECODE=1
        build:
            dockerfile: docker/app/Dockerfile
            context: .
        entrypoint: pytest
        volumes:
            - ./ThoughtsBackend:/opt/code
```

This section defines a service called `test-sqlite`. The build defines the Dockerfile to use and the context, in the same way as we'd do with a `docker build` command. `docker-compose` automatically sets the name.

We can build the container with the following command:

```
$ docker-compose build test-sqlite
Building test-sqlite
...
Successfully built 8751a4a870d9
Successfully tagged ch3_test-sqlite:latest
```

`entrypoint` specifies the command to run, in this case, running the tests through the `pytest` command.

There are some differences between the command and the `entrypoint`, which both execute a command. The most relevant ones are that `command` is easier to overwrite and `entrypoint` appends any extra arguments at the end.

To run the container, call the `run` command:

```
$ docker-compose run test-sqlite
=================== test session starts ====================
platform linux -- Python 3.6.8, pytest-4.5.0, py-1.8.0, pluggy-0.12.0 --
/opt/venv/bin/python3
cachedir: .pytest_cache
rootdir: /opt/code, inifile: pytest.ini
plugins: flask-0.14.0
collected 17 items

tests/test_thoughts.py::test_create_me_thought PASSED [ 5%]
...
tests/test_token_validation.py::test_valid_token_header PASSED [100%]

========== 17 passed, 177 warnings in 1.25 seconds ============
$
```

You can append `pytest` arguments that will be passed over to the internal `entrypoint`. For example, to run tests that match the *validation* string, run the following command:

```
$ docker-compose run test-sqlite -k validation
...
===== 9 passed, 8 deselected, 13 warnings in 0.30 seconds =======
$
```

There are two extra details: the current code is mounted through a volume and overwrites the code in the container. See how the current code in `./ThoughtsBackend` is mounted in the position of the code in the container, `/opt/code`. This is very handy for the development, as it will avoid having to rebuild the container each time a change is made.

This also means that any write in the mounted directory hierarchy will be saved in your local filesystem. For example, the `./ThoughtsBackend/db.sqlite3` database file allows you to use it for testing. It will also store generated `pyc` files.

The generation of the `db.sqlite3` file can create permission problems in some operating systems. If that's the case, delete it to be regenerated and/or allow it to read and write to all users with `chmod 666 ./ThoughtsBackend/db.sqlite3`.

That's why we use the `environment` option to pass a `PYTHONDONTWRITEBYTECODE=1` environment variable. This stops Python from creating `pyc` files.

While SQLite is good for testing, we need to create a better structure reflective of the deployment and to configure the access to the database to be able to deploy the server.

Creating a PostgreSQL database container

We need to test our code against a PostgreSQL database. This is the database that we will be deploying the code in production against.

While the abstraction layer in SQLAlchemy aims to reduce the differences, there are some differences in the behavior of the databases.

For example, in `/thoughts_backend/api_namespace.py`, the following line is case-insensitive, which is the behavior that we want:

```
query = (query.filter(ThoughtModel.text.contains(search_param)))
```

Translating that to PostgreSQL, it is case-sensitive, which requires you to check it. This would be a bug in production if testing with SQLite and running in PostgreSQL.

The replaced code, using `ilike` for the expected behavior, is as follows:

```
param = f'%{search_param}%'
query = (query.filter(ThoughtModel.text.ilike(param)))
```

We kept the old code in a comment to show this issue.

To create a database container, we need to define the corresponding Dockerfile. We store all the files in the `docker/db/` subdirectory. Let's take a look at Dockerfile and its different parts. The whole file can be found on GitHub (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter03/docker/db/Dockerfile`). This Dockerfile can be divided into the following stages:

1. Using the `ARG` keyword, define the basic PostgreSQL configuration such as the name of the database, user, and password. They get set in environment variables so that the PostgreSQL commands can use them.

These commands are for local development only. They'll need to match with the environment set up. The `ARG` keyword defines a parameter for Dockerfile at build time. We'll see how they are set up as input parameters in the `docker-compose.yaml` file.

The `ARG` elements are also defined as `ENV` variables, so we keep them defined as environment variables:

```
# This Dockerfile is for localdev purposes only, so it won't be
# optimised for size
FROM alpine:3.9

# Add the proper env variables for init the db
ARG POSTGRES_DB
ENV POSTGRES_DB $POSTGRES_DB
ARG POSTGRES_USER
ENV POSTGRES_USER $POSTGRES_USER
ARG POSTGRES_PASSWORD
ENV POSTGRES_PASSWORD $POSTGRES_PASSWORD
ARG POSTGRES_PORT
ENV LANG en_US.utf8
EXPOSE $POSTGRES_PORT

# For usage in startup
ENV POSTGRES_HOST localhost
ENV DATABASE_ENGINE POSTGRESQL
# Store the data inside the container, as we don't care for
# persistence
RUN mkdir -p /opt/data
ENV PGDATA /opt/data
```

2. Install the `postgresql` package and all its dependencies, such as Python 3 and its compilers. We will need them to be able to run the application code:

```
RUN apk update
RUN apk add bash curl su-exec python3
RUN apk add postgresql postgresql-contrib postgresql-dev
RUN apk add python3-dev build-base linux-headers gcc libffi-dev
```

3. Install and run the `postgres-setup.sh` script:

```
# Adding our code
WORKDIR /opt/code

RUN mkdir -p /opt/code/db
# Add postgres setup
ADD ./docker/db/postgres-setup.sh /opt/code/db/
RUN /opt/code/db/postgres-setup.sh
```

This initializes the database, setting the correct user, password, and so on. Note that this doesn't create the specific tables for our application yet.

As part of our initialization, we create the data files inside the container. This means that the data won't persist after the container stops. This is a good thing for testing, but, if you want to access the data for debug purposes, remember to keep the container up.

4. Install the requirements for our application and specific commands to run in the database container:

```
## Install our code to prepare the DB
ADD ./ThoughtsBackend/requirements.txt /opt/code

RUN pip3 install -r requirements.txt
```

5. Copy the application code and database commands stored in `docker/db`. Run the `prepare_db.sh` script, which creates the application database structure. In our case, it sets up the `thoughts` table:

```
## Need to import all the code, due dependencies to initialize the
DB
ADD ./ThoughtsBackend/ /opt/code/
# Add all DB commands
ADD ./docker/db/* /opt/code/db/

## get the db ready
RUN /opt/code/db/prepare_db.sh
```

This script first starts the PostgreSQL database running in the background, then calls `init_db.py`, and then gracefully stops the database.

Keep in mind that, in each of the steps of Dockerfile, in order to access the database, it needs to be running, but it will also be stopped at the end of each step. In order to avoid corruption of the data or the abrupt killing of the process, be sure to use the `stop_postgres.sh` script until the end. Though PostgreSQL will normally recover for an abruptly stopped database, it will slow the startup time.

6. To start the database in operation, the CMD is just the `postgres` command. It needs to run with the `postgres` user:

```
# Start the database in normal operation
USER postgres
CMD ["postgres"]
```

To run the database service, we need to set it up as part of the `docker-compose` file:

```
db:
    build:
        context: .
        dockerfile: ./docker/db/Dockerfile
        args:
            # These values should be in sync with environment
            # for development. If you change them, you'll
            # need to rebuild the container
            - POSTGRES_DB=thoughts
            - POSTGRES_USER=postgres
            - POSTGRES_PASSWORD=somepassword
            - POSTGRES_PORT=5432
    ports:
        - "5432:5432"
```

Note that the `args` parameter will set up the `ARG` values during the build. We also route the PostgreSQL port to allow access to the database.

You can now build and start the server:

```
$ docker-compose up build
$ docker-compose up db
Creating ch3_db_1 ... done
Attaching to ch3_db_1
...
db_1 | 2019-06-02 13:55:38.934 UTC [1] LOG: database system is ready to
accept connections
```

In a different Terminal, you can use a PostgreSQL client to access the database. I recommend the fantastic `pgcli`. You can check out its documentation (`https://www.pgcli.com/`).

You can use also the official `psql` client or any other PostgreSQL client of your preference. The documentation for the default client can be found here: `https://www.postgresql.org/docs/current/app-psql.html`.

Here, we use the `PGPASSWORD` environment variable to show that the password is the previously configured one:

```
$ PGPASSWORD=somepassword pgcli -h localhost -U postgres thoughts
Server: PostgreSQL 11.3
Version: 2.0.2
Chat: https://gitter.im/dbcli/pgcli
Mail: https://groups.google.com/forum/#!forum/pgcli
Home: http://pgcli.com
postgres@localhost:thoughts> select * from thought_model
+------+------------+--------+-------------+
|  id  |  username  |  text  |  timestamp  |
|------+------------+--------+-------------|
+------+------------+--------+-------------+
SELECT 0
Time: 0.016s
```

Being able to access the database is useful for debugging purposes.

Configuring your service

We can configure the service to use environment variables to change the behavior. For containers, this is a fantastic alternative to using configuration files, as it allows immutable containers that get their configuration injected. This is in line with the Twelve-Factor App (`https://12factor.net/config`) principles and allows for good separation between code and configuration, and the setting up of the different deploys that the code might be used for.

One of the advantages that we'll look at later with the use of Kubernetes is creating new environments on-demand, which can be tweaked for testing purposes or tailored for development or demo. Being able to quickly change all the configuration by injecting the proper environment makes this operation very easy and straightforward. It also allows you to enable or disable features, if properly configured, which helps the enablement of features on launch day, with no code rollout.

This allows the configuration of the database to connect to, so we can choose between the SQLite backend or PostgreSQL.

Configuring the system is not limited to open variables, though. Environment variables will be used later in the book for storing secrets. Note that a secret needs to be available inside the container.

We will configure the tests to access our newly created database container. To do that, we first need the ability to choose between either SQLite or PostgreSQL through configuration. Check out the `./ThoughtsBackend/thoughts_backend/db.py` file:

```
import os
from pathlib import Path
from flask_sqlalchemy import SQLAlchemy

DATABASE_ENGINE = os.environ.get('DATABASE_ENGINE', 'SQLITE')

if DATABASE_ENGINE == 'SQLITE':
    dir_path = Path(os.path.dirname(os.path.realpath(__file__)))
    path = dir_path / '..'

    # Database initialisation
    FILE_PATH = f'{path}/db.sqlite3'
    DB_URI = 'sqlite+pysqlite:///{file_path}'
    db_config = {
        'SQLALCHEMY_DATABASE_URI': DB_URI.format(file_path=FILE_PATH),
        'SQLALCHEMY_TRACK_MODIFICATIONS': False,
    }

elif DATABASE_ENGINE == 'POSTGRESQL':
    db_params = {
        'host': os.environ['POSTGRES_HOST'],
        'database': os.environ['POSTGRES_DB'],
        'user': os.environ['POSTGRES_USER'],
        'pwd': os.environ['POSTGRES_PASSWORD'],
        'port': os.environ['POSTGRES_PORT'],
    }
```

```
        DB_URI = 'postgresql://{user}:{pwd}@{host}:{port}/{database}'
        db_config = {
            'SQLALCHEMY_DATABASE_URI': DB_URI.format(**db_params),
            'SQLALCHEMY_TRACK_MODIFICATIONS': False,
        }

    else:
        raise Exception('Incorrect DATABASE_ENGINE')

    db = SQLAlchemy()
```

When using the `DATABASE_ENGINE` environment variable set to `POSTGRESQL`, it will configure it properly. Other environment variables will need to be correct; that is, if the database engine is set to PostgreSQL, the `POSTGRES_HOST` variable needs to be set up.

Environment variables can be stored individually in the `docker-compose.yaml` file, but it's more convenient to store multiple ones in a file. Let's take a look at `environment.env`:

```
DATABASE_ENGINE=POSTGRESQL
POSTGRES_DB=thoughts
POSTGRES_USER=postgres
POSTGRES_PASSWORD=somepassword
POSTGRES_PORT=5432
POSTGRES_HOST=db
```

Note that the definition of users, and so on is in line with the arguments to create Dockerfile for testing. `POSTGRES_HOST` is defined as `db`, which is the name of the service.

Inside the Docker cluster created for `docker-compose`, you can refer to services by their names. This will be directed by the internal DNS to the proper container, as a shortcut. This allows easy communication between services, as they can configure their access very easily by name. Note that this connection is only valid inside the cluster, for communication between containers.

Our testing service using the PostgreSQL container then gets defined as follows:

```
    test-postgresql:
        env_file: environment.env
        environment:
            - PYTHONDONTWRITEBYTECODE=1
        build:
            dockerfile: docker/app/Dockerfile
            context: .
        entrypoint: pytest
        depends_on:
            - db
        volumes:
            - ./ThoughtsBackend:/opt/code
```

This is very similar to the `test-sqlite` service, but it adds the environment configuration in `environment.env` and adds a dependency on `db`. This means that `docker-compose` will start the `db` service, if not present.

You can now run the tests against the PostgreSQL database:

```
$ docker-compose run test-postgresql
Starting ch3_db_1 ... done
============== test session starts ====================
platform linux -- Python 3.6.8, pytest-4.6.0, py-1.8.0, pluggy-0.12.0 --
/opt/venv/bin/python3
cachedir: .pytest_cache
rootdir: /opt/code, inifile: pytest.ini
plugins: flask-0.14.0
collected 17 items

tests/test_thoughts.py::test_create_me_thought PASSED [ 5%]
...
tests/test_token_validation.py::test_valid_token_header PASSED [100%]

===== 17 passed, 177 warnings in 2.14 seconds ===
$
```

This environment file will be useful for any service that needs to connect to the database, such as deploying the service locally.

Deploying the Docker service locally

With all these elements, we can create the service to locally deploy the Thoughts service:

```
server:
   env_file: environment.env
   image: thoughts_server
   build:
       context: .
       dockerfile: docker/app/Dockerfile
   ports:
       - "8000:8000"
   depends_on:
       - db
```

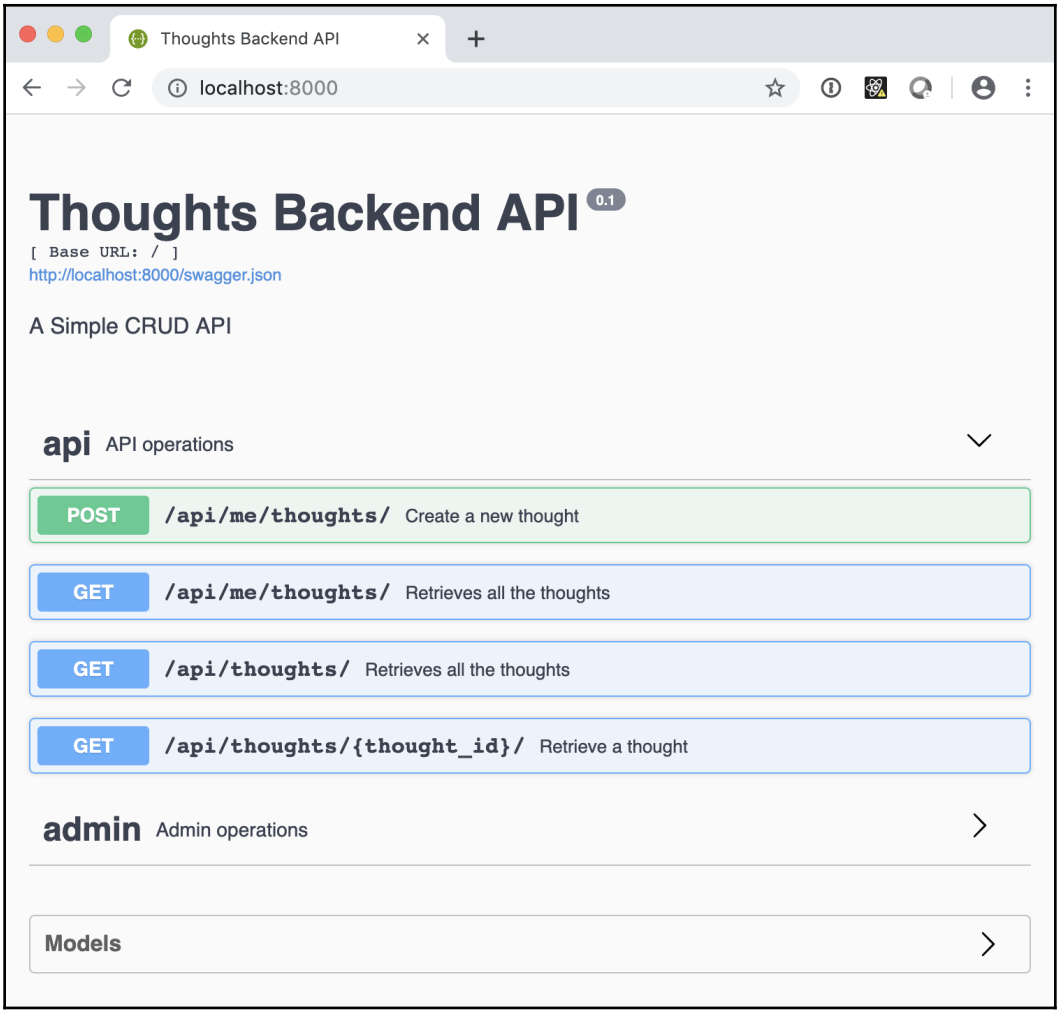
We need to be sure to add the dependency of the `db` database service. We also bound the internal port so that we can access it locally.

We start the service with the `up` command. There are some differences between the `up` and the `run` commands, but the main one is that `run` is for single commands that start and stop, while `up` is designed for services. For example, `run` creates an interactive Terminal, which displays colors, and `up` shows the standard output as logs, including the time when they were generated, accepts the `-d` flag to run in the background, and so on. Using one instead of the other is normally okay, however, `up` exposes ports and allows other containers and services to connect, while `run` does not.

We can start the service now with the following commands:

```
$ docker-compose up server
Creating network "ch3_default" with the default driver
Creating ch3_db_1 ... done
Creating ch3_server_1 ... done
Attaching to ch3_server_1
server_1 | [uWSGI] getting INI configuration from /opt/uwsgi/uwsgi.ini
server_1 | *** Starting uWSGI 2.0.18 (64bit) on [Sun Jun 2
...
server_1 | spawned uWSGI master process (pid: 6)
server_1 | spawned uWSGI worker 1 (pid: 7, cores: 1)
server_1 | spawned uWSGI http 1 (pid: 8)
```

Now access the service in `localhost:8000` in a browser:



You can see the logs in the Terminal. Hitting *Ctrl + C* will stop the server. The service can also be started using the `-d` flag, to detach the Terminal and run in daemon mode:

```
$ docker-compose up -d server
Creating network "ch3_default" with the default driver
Creating ch3_db_1 ... done
Creating ch3_server_1 ... done
$
```

Check the running services, their current state, and open ports with `docker-compose ps`:

```
$ docker-compose ps
    Name Command State Ports
------------------------------------------------------------------------------
---
ch3_db_1 postgres Up 0.0.0.0:5432->5432/tcp
ch3_server_1 /bin/sh /opt/uwsgi/start_s ... Up 0.0.0.0:8000->8000/tcp
```

As we've seen before, we can directly access the database and run raw SQL commands in it. This can be useful for debugging problems or conducting experiments:

```
$ PGPASSWORD=somepassword pgcli -h localhost -U postgres thoughts
Server: PostgreSQL 11.3
Version: 2.0.2

postgres@localhost:thoughts>
INSERT INTO thought_model (username, text, timestamp)
VALUES ('peterparker', 'A great power carries a great
 responsability', now());

INSERT 0 1
Time: 0.014s
postgres@localhost:thoughts>
```

Now the thought is available through the following API:

```
$ curl http://localhost:8000/api/thoughts/
[{"id": 1, "username": "peterparker", "text": "A great power carries a
great responsability", "timestamp": "2019-06-02T19:44:34.384178"}]
```

If you need to see the logs in detach mode, you can use the `docker-compose logs <optional: service>` command:

```
$ docker-compose logs server
Attaching to ch3_server_1
server_1 | [uWSGI] getting INI configuration from /opt/uwsgi/uwsgi.ini
server_1 | *** Starting uWSGI 2.0.18 (64bit) on [Sun Jun 2 19:44:15 2019]
***
server_1 | compiled with version: 8.3.0 on 02 June 2019 11:00:48
...
server_1 | [pid: 7|app: 0|req: 2/2] 172.27.0.1 () {28 vars in 321 bytes}
[Sun Jun 2 19:44:41 2019] GET /api/thoughts/ => generated 138 bytes in 4
msecs (HTTP/1.1 200) 2 headers in 72 bytes (1 switches on core 0)
```

To totally stop the cluster, call `docker-compose down`:

```
$ docker-compose down
Stopping ch3_server_1 ... done
Stopping ch3_db_1 ... done
Removing ch3_server_1 ... done
Removing ch3_db_1 ... done
Removing network ch3_default
```

This stops all the containers.

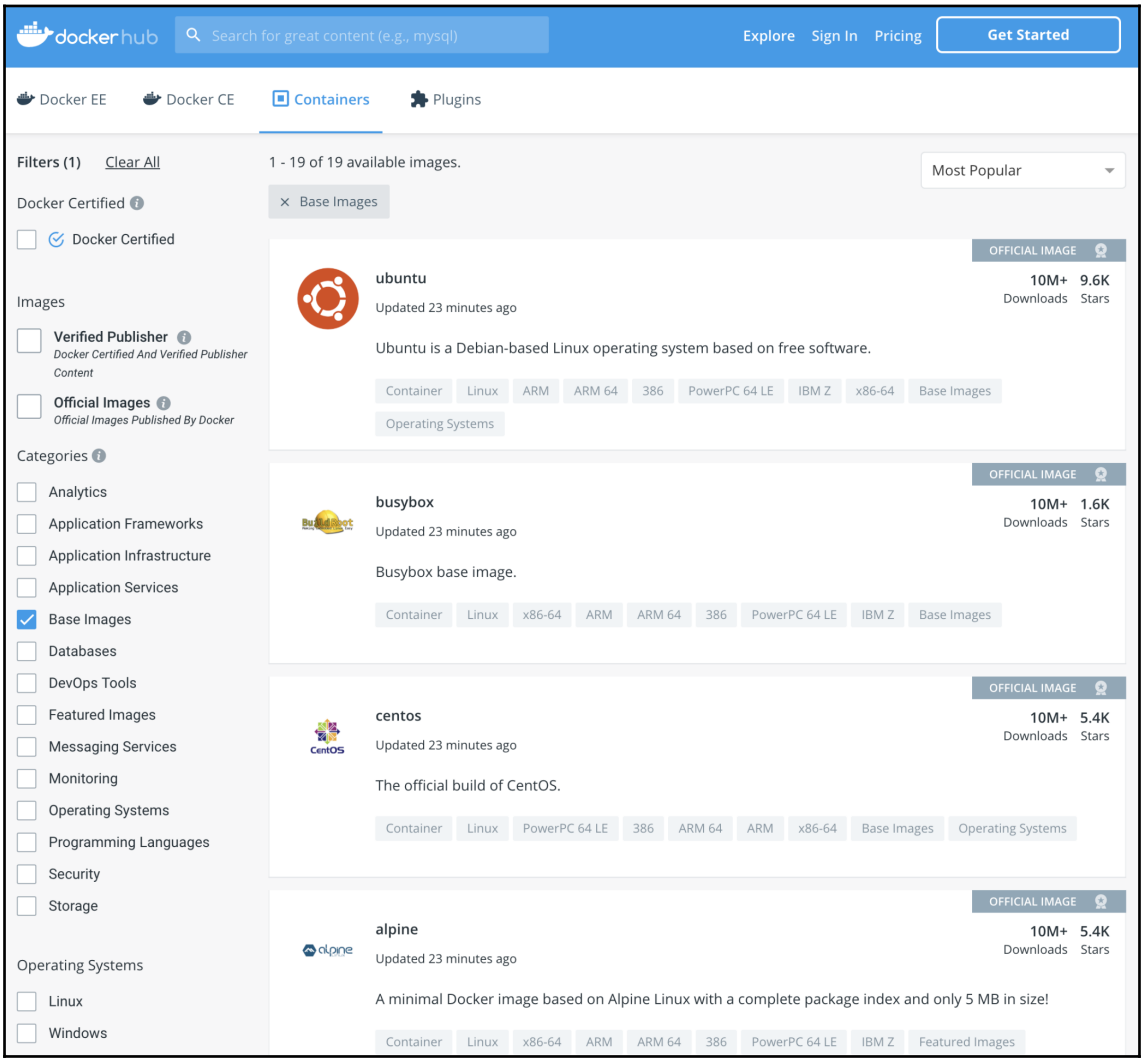
Pushing your Docker image to a remote registry

All the operations that we've seen work with our local Docker repository. Given the structure of Docker images and the fact that each layer can be worked on independently, they are easy to upload and share. To do so, we need to use a remote repository, or registry in Docker terminology, that will accept images pushed to it, and allow images to be pulled from it.

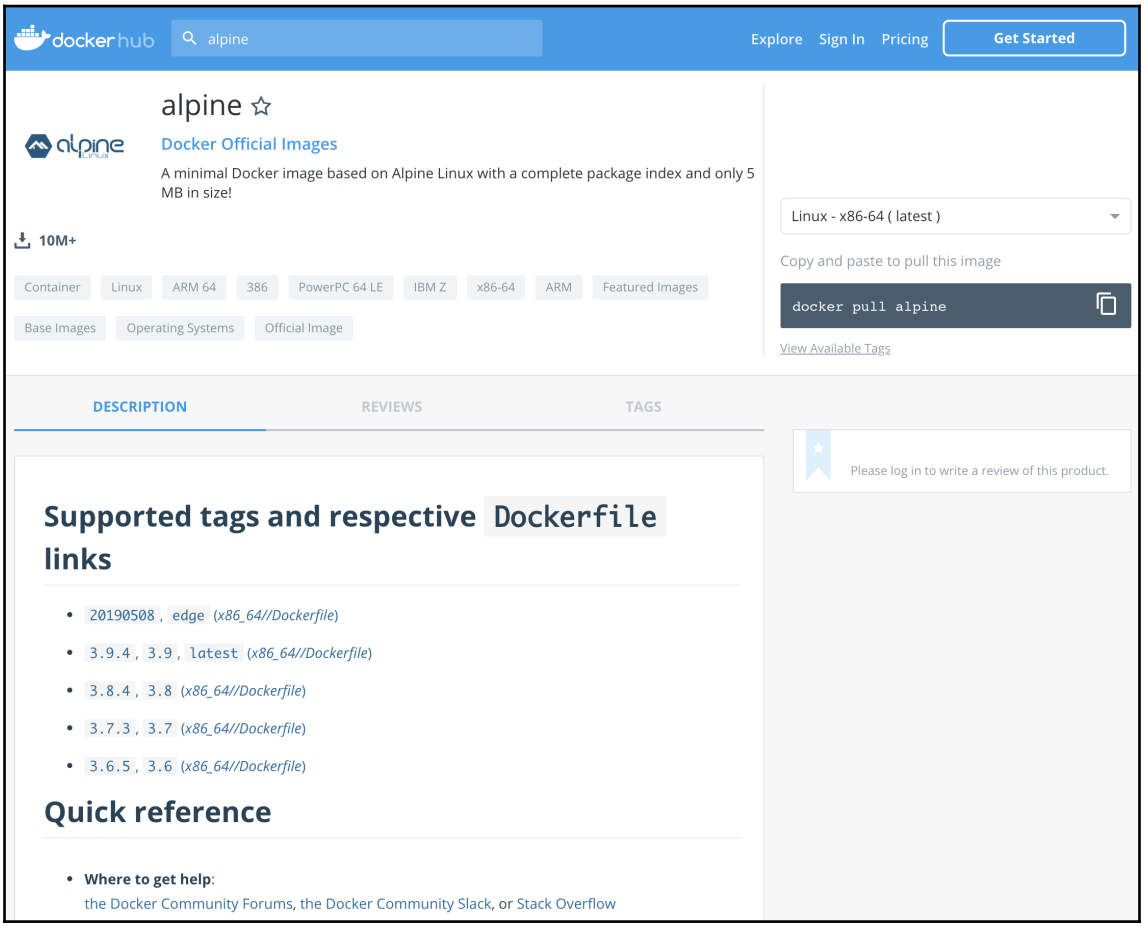
The structure of a Docker image is composed of each of the layers. Each of them can be pushed independently, as long as the registry contains the layer it depends on. This saves space if the previous layers are already present, as they will be stored only once.

Obtaining public images from Docker Hub

The default registry is Docker Hub. This is configured by default, and it serves as the main source of public images. You can access it freely in `https://hub.docker.com/` and search for available images to base your images on:



Each image has information about the way to use it and the tags that are available. You don't need to download the images independently, just to use the name of the image or run a `docker pull` command. Docker will automatically pull from Docker Hub if no other registry is specified:



The name of the image is also the one to use in our `FROM` command in Dockerfiles.

Docker is a fantastic way of distributing a tool. It's very common right now for an open source tool to have an official image in Docker Hub that can be downloaded and started in a standalone model, standardizing the access.

This can be used either for a quick demo, for something such as Ghost—`https://hub.docker.com/_/ghost` (a blogging platform), or a Redis (`https://hub.docker.com/_/redis`) instance to act as cache with minimal work. Try to run the Ghost example locally.

Using tags

Tags are descriptors to label different versions of the same image. There's an image, `alpine:3.9`, and another, `alpine:3.8`. There are also official images of Python for different interpreters (3.6, 3.7, 2.7, and so on), but other than versions, the interpreters may refer to ways the image is created.

For example, these images have the same effect. The first one is a full image containing a Python 3.7 interpreter:

```
$ docker run -it python:3.7
Python 3.7.3 (default, May 8 2019, 05:28:42)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The second one also has a Python 3.7 interpreter. Note the `slim` change in the name:

```
$ docker run -it python:3.7-slim
Python 3.7.3 (default, May 8 2019, 05:31:59)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

However, the sizes of the images are quite different:

```
$ docker images | grep python
python 3.7-slim ca7f9e245002 4 weeks ago 143MB
python 3.7      a4cc999cf2aa 4 weeks ago 929MB
```

Any build uses the `latest` tag automatically if another tag is not specified.

Keep in mind that tags can be overwritten. This may be confusing, given some of the similarities between the way Docker and Git work, as the term "tag" in Git means something that can't change. A tag in Docker is similar to a branch in Git.

A single image can be tagged multiple times, with different tags. For example, the `latest` tag can also be version `v1.5`:

```
$ docker tag thoughts-backend:latest thoughts-backend:v1.5
$ docker images
REPOSITORY       TAG    IMAGE ID     CREATED    SIZE
thoughts-backend latest c7a8499623e7 5 min ago 144MB
thoughts-backend v1.5   c7a8499623e7 5 min ago 144MB
```

Note how `image id` is the same. Using tags allows you to label specific images, so we know they are ready to deploy or give them some kind of significance.

Pushing into a registry

Once we have our image tagged, we can push it to a shared registry so that it's available for other services to use.

It is possible to deploy your own Docker registry, but, unless strictly necessary, it's better to avoid it. There are cloud providers that allow you to create your own registry, either public or private, and even in your own private cloud network. If you want to make your image available, the best alternative is Docker Hub, as it's the standard and it will be most accessible. In this chapter, we will create one here, but we'll explore other options later in the book.

It's worth saying it again: maintaining your own Docker registry is much more expensive than using a provider one. Commercial prices for registries, unless you require a lot of repos will be in the range of tens of dollars per month, and there are options from well-known cloud providers such as AWS, Azure, and Google Cloud.

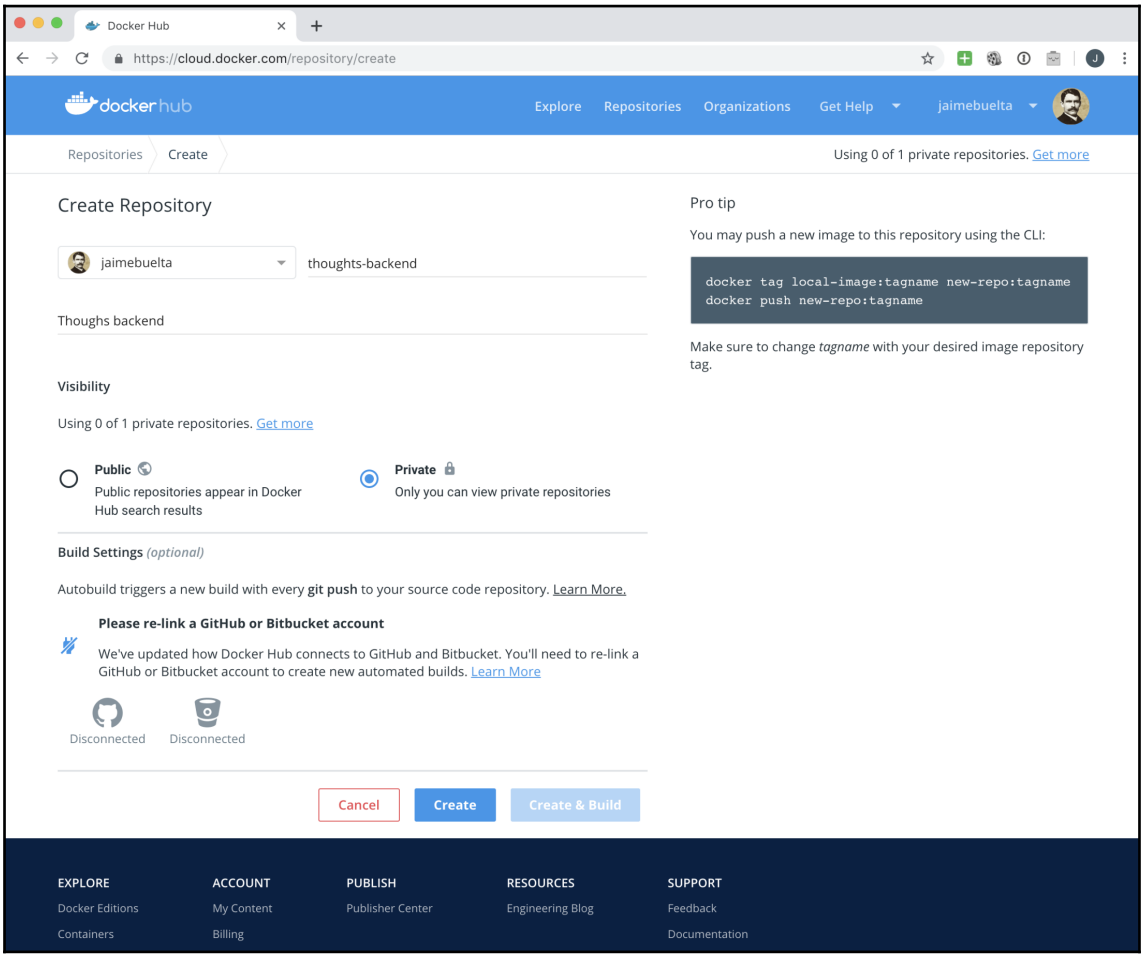
Unless you really need to, avoid using your own registry.

We will create a new repo in the Docker Hub registry. You can create a private repo for free, and as many public ones as you want. You need to create a new user, which was probably the case when downloading Docker.

A repo, in Docker terms, is a set of images with different tags; for example, all the tags of `thoughts-backend`. This is different from the registry, which is a server that contains several repos.

In more informal terms, it's common to refer to registries as *repos* and to repos as *images*, though, speaking purely, an image is unique and may be a tag (or not).

Then, you can create a new repo as follows:



Once the repo is created, we need to tag our image accordingly. This means that it should include the username in Docker Hub to identify the repo. An alternative is to name the image directly with the username included:

```
$ docker tag thoughts-backend:latest jaimebuelta/thoughts-backend:latest
```

To be able to access the repo, we need to log into Docker with our username and password in Docker Hub:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: jaimebuelta
Password:
Login Succeeded
```

Once logged in, you can push your image:

```
$ docker push jaimebuelta/thoughts-backend:latest
The push refers to repository [docker.io/jaimebuelta/thoughts-backend]
1ebb4000a299: Pushed
669047e32cec: Pushed
6f7246363f55: Pushed
ac1d27280799: Pushed
c43bb774a4bb: Pushed
992e49acee35: Pushed
11c1b6dd59b3: Pushed
7113f6aae2a4: Pushed
5275897866cf: Pushed
bcf2f368fe23: Mounted from library/alpine
latest: digest:
sha256:f1463646b5a8dec3531842354d643f3d5d62a15cc658ac4a2bdbc2ecaf6bb145
size: 2404
```

You can now share the image and pull it from anywhere, given that the local Docker is properly logged. When we deploy a production cluster, we need to be sure that the Docker server executing it is capable of accessing the registry and that it's properly logged.

Summary

In this chapter, we learned how to use Docker commands to create and operate containers. We learned most of the commonly used Docker commands, such as `build`, `run`, `exec`, `ps`, `images`, `tag`, and `push`.

We saw how to build a web service container, including the preparation of configuration files, how to structure a Dockerfile, and how to make our images as small as possible. We also covered how to use `docker-compose` to operate locally and, through a `docker-compose.yaml` file, connect different containers running in a cluster configuration. This included creating a database container that allows testing much closer to what the production deployment will be, using the same tools.

We saw how to use environment variables to configure our service and how to inject them through `docker-compose` configuration to allow different modes, such as testing.

Finally, we analyzed how to use a registry to share our images, and how to tag them adequately and allow moving them out from local development, ready to be used in a deployment.

In the next chapter, we will see how to leverage the created containers and actions to run tests automatically and make automated tools do the heavy lifting for us to be sure our code is always high-quality!

Questions

1. What does the `FROM` keyword do in a Dockerfile?
2. How would you start a container with its predefined command?
3. Why won't creating a step to remove files in a Dockerfile make a smaller image?
4. Can you describe how a multistage Docker build works?
5. What is the difference between the `run` and `exec` commands?
6. When should we use the `-it` flags when using the `run` and `exec` commands?
7. Do you know any alternatives to uWSGI to serve Python web applications?
8. What is `docker-compose` used for?
9. Can you describe what a Docker tag is?
10. Why is it necessary to push images to a remote registry?

Further reading

To further your knowledge of Docker and containers, you can check out the *Mastering Docker – Third Edition* book (`https://www.packtpub.com/eu/virtualization-and-cloud/mastering-docker-third-edition`). For tweaking containers and learning how to make your applications more performant, see *Docker High Performance - Second Edition* (`https://www.packtpub.com/eu/networking-and-servers/docker-high-performance-second-edition`), which covers a lot of techniques for analyzing and discovering performance problems.

4
Creating a Pipeline and Workflow

A pipeline that runs automatically in a workflow, through different stages, will detect problems early and help your team collaborate in the most efficient way.

In this chapter, we will follow continuous integration practices, running the pipeline automatically and on every change, to be sure that all our code follows high quality standards, and that it runs and passes all tests. We'll also get a container ready to go to production.

We will see how to leverage tools such as GitHub and Travis CI to create images with minimal intervention.

In this chapter, we will be covering the following topics:

- Understanding continuous integration practices
- Configuring Travis CI
- Configuring GitHub
- Pushing Docker images from Travis CI

By the end of the chapter, you'll know how to automatically run tests on every code change and how to create a safety net that will allow you to develop faster and more efficiently.

Technical requirements

You require a GitHub account and need to be the owner of the project you'll set up for continuous integration. We will create a Travis CI account as part of this chapter.

You can checkout the full code referred to in this chapter in the `Chapter04` subdirectory from GitHub (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter04`). The file ending with `.travis.yml` is in the root directory (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/.travis.yml`).

Understanding continuous integration practices

Continuous integration (usually abbreviated as **CI**) is a series of software engineering practices that ensure that code is always in a working state.

The term continuous integration comes from historically having to integrate software frequently, often multiple times a day. This arose from the fact that developers worked with local code that was not necessarily joined with other people's code automatically. Nowadays, using a source-control versioning software such as Git makes some of the elements automatically available.

Continuous integration emphasizes on having potentially releasable code at all times. This makes releases possible very often, with small increments of code.

Making more releases more often actually generates an increase in the quality of each release. More deployments also mean that each deployment is smaller, reducing the possibility of a big problem. Even if it sounds counterintuitive, faster deployment is highly correlated with higher quality in deployments and fewer production problems.

The objective here is to be able to increase the deployment speed. But for that, we need to be sure to build a good safety net that checks (automatically) that what we're doing is safe to release. That's where all the CI practices come into play.

It is quite possible, after setting all the processes and infrastructure in place, to implement releases multiple times a day (assuming that the code is generated fast enough). It can take a while to get there, but be sure to take your time to understand the process and produce all the necessary tools to be certain that you gain speed without sacrificing stability. And, trust me, it is totally achievable!

Producing automated builds

The core element in CI is generating automated builds integrated with a source control system. A software build is a process that (starting from the source code) performs a series of actions and produces an output. If the project is written in a compiled language, the output will typically be the compiled program.

If we want to have quality software, then part of the build involves checking that the produced code follows code standards. If the code doesn't follow those standards, then the build will return an error.

A common way of describing errors on a build is to say that the *build is broken*. A build can break in different ways, and some kinds of error may stop it early (such as a compilation error before running tests) or we can continue to detect further issues (such as running all tests to return all possible errors).

Some examples of steps that can be a part of the build are as follows:

- Compiling the code.

Python usually doesn't need to be compiled, but it might be required if you use C extensions (modules written in C and imported from Python: `https://docs.python.org/3/extending/`) or tools such as Cython (`https://cython.org/`).

- Running unit tests
- Running static code analysis tools
- Building one or more containers
- Checking dependencies for known vulnerabilities with a tool such as Safety (`https://pyup.io/safety/`)
- Generating a binary or source package for distribution. For example, RPM (`https://rpm.org/`), Debian packages (`https://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics`), and so on
- Running other kinds of tests
- Generating reports, diagrams, or other assets from the code

Anything that can run automatically can be a part of a build. A local build can be generated at any time, even with code that's still in progress. This is important for debugging and solving issues. But automated builds will run against each individual commit, and not at any intermediate stage. This makes it very explicit to check what code is expected to run in production and what code is still work in progress.

Note that a single commit may still be work in progress, but it will be worth committing anyway. Maybe it's a single step toward a feature, more than one person is working on the same part of the code, or it's work spread over several days and the code gets pushed at the end of the day. No matter, each commit is a reproducible step that can be built and checked whether the build is successful or not.

Running the build for each commit detects problems very quickly. If commits are small, a breaking change is easy to pinpoint. It also makes it easy to revert changes that break the build and go back to known working code.

Knowing the advantages of using Docker for builds

One of the main traditional problems with builds was having an adequate build environment with all the dependencies needed to run the full build. This could include things such as the compiler, the test framework to run the tests, any static analysis tools, and the package manager. A discrepancy in versions could also produce errors.

As we've seen before, Docker is a fantastic way of encapsulating our software. It allows us to create an image that contains both our code and all tools that are able to proceed through all the steps.

In the previous chapter, we saw how to run unit tests in a single command, based on a build image. The image itself can run its own unit tests. This abstracts the test environment and explicitly defines it. The only dependency necessary here is to have Docker installed.

Keep in mind that a single build could generate multiple images and make them work in coordination. We saw how to run unit tests in the previous chapter—by generating service image and a database image—but there are more possible usages. For example, you could check the test running on two different operating systems, creating two images from each of the operating systems or different Python interpreter versions, and checking whether the tests pass in all of them.

The usage of Docker images allows for standardization in all environments. We can locally run images in a development environment, using the same commands that we did in our automated environment. This streamlines finding bugs and problems, as it creates the same environment, including an encapsulated operating system, everywhere the build is run.

Do not underestimate this element. Before that, a developer working on a laptop running Ubuntu and keen to run code to be deployed in CentOS needed to install a **Virtual Machine** (**VM**) and follow steps to have an environment similar to the one in production. But invariably, the local VM would deviate as it was difficult to keep every developer's local VM in sync with the one in production; also, any automated-build tool might also have requirements, such as not supporting an old version of CentOS running in production.

To make things worse, sometimes different projects were installed on the same VM, to avoid having one VM per project, and that may cause compatibility problems.

Docker massively simplifies this problem, in part forcing you to explicitly declare what the dependencies are, and reducing the surface actually required to run our code.

Note that we don't necessarily need to create a single step that runs the whole build; it could be several Docker commands, even making use of different images. But the requirement is that they are all contained in Docker, the only software required to run it.

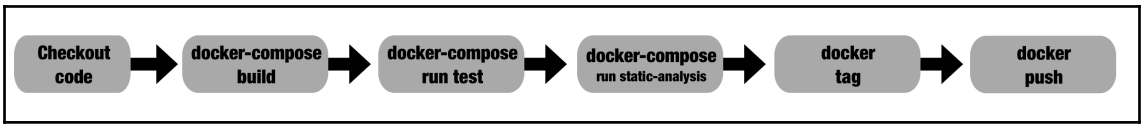
The main product of a build using Docker is Docker image or images. We will need to properly tag them, but only if the build is successful.

Leveraging the pipeline concept

CI tools help to clarify how a build should proceed and work around the concept of a pipeline. A pipeline is a collection of stages. If any of them are not successful, the pipeline stops.

Each stage in the pipeline can produce elements that could be used at later stages or are available as the final product of the full build. These final elements are known as artifacts.

Let's look at an example of a pipeline:

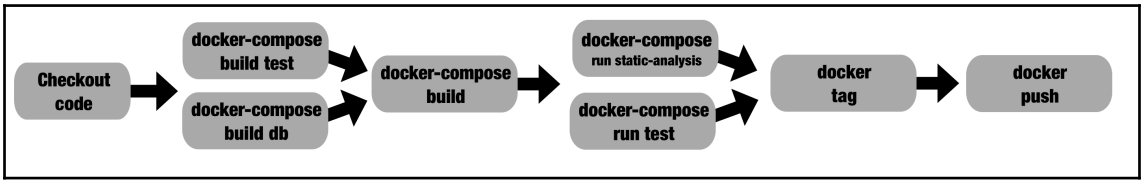


The first stage pulls the latest commit from the source control system. Then, we build all the containers and run both tests and the static analysis. If all has been successful, we tag the resulting `server` container and push it to the registry.

The order in which these stages run should be oriented at detecting problems as quickly as possible to give quick feedback. For example, if the `static-analysis` stage is much faster than the `test` stage, putting the analysis stage first will make a failing build finish earlier. Be aware of which parts can be executed earlier to reduce the feedback time.

CI tools normally allow great configuration in pipelines, including the possibility of running different stages in parallel. To be able to run stages in parallel, they need to be able to be parallelizable, meaning that they should not change the same elements.

If the chosen CI tool allows running stages in parallel, the pipeline could be defined as follows:



Note that we build the database and the test images in parallel. The next stage builds the rest of the elements, which are already available in the cache, so it will be very quick. Both the tests and the static analysis can run in parallel, in two different containers.

This may speed up complex builds.

Be sure to validate that the amount of time taken reduces. There are cases where the time taken will be very similar. For example, static analysis could be very fast or the hardware you run it on may be not powerful enough to build things in parallel, making the time taken to build in parallel and sequentially very similar. So, always validate your assumptions.

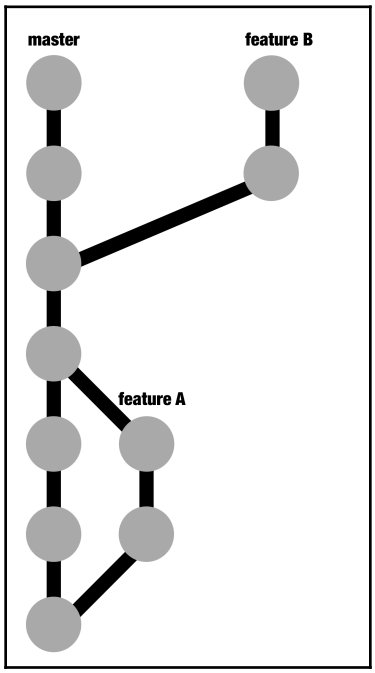
The pipeline is described in a script specific to the Travis CI tool. We'll look at an example with Travis CI later.

Branching, merging, and ensuring a clear main build

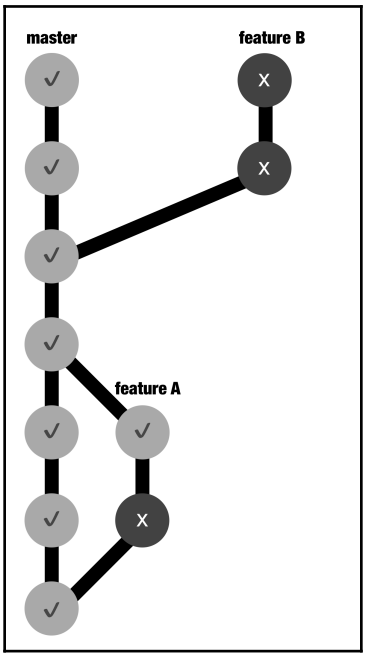
When do we run a build? Every time a commit is pushed. But every result is not the same. When dealing with a source control system such as Git, we typically have two kinds of branches:

- One main branch
- Feature branches

They implement a particular feature or bugfix, which will be merged into the main branch when ready, as is shown in the following figure:



In this example, we see how the main branch (**master**) is branched to develop **feature A**. **Feature A** is introduced briefly after that. There is a **feature B** that hasn't been merged yet since it's not ready. With the extra information on what builds have been successful or not, we can know when it is safe to merge a feature branch into the main branch:



Breakage in a feature branch that is not yet merged is not great, but while it is work in progress, it is expected to happen. Meanwhile, a breakage in the main branch is an event that should be fixed as soon as possible. If the main branch is in good shape, that means that it is potentially releasable.

GitHub has a model for this: pull requests. We will configure pull requests to automatically check whether the build has passed and avoided merging or not. If we force any feature branch to also be up-to-date with the main branch before merging back, the main branch ends up being very stable.

For dealing with branches in Git to define releases, the most popular model is Git-flow, defined in this influential blog post (`https://nvie.com/posts/a-successful-git-branching-model/`). The following CI practices allow simplify things a bit and don't deal with elements such as release branches. This blog post is a highly recommended read.

Having an uninterrupted line of successful builds in the main branch is also very helpful to develop a sense of stability and quality in the project. If main branch breakages are very rare, confidence in creating a new release with the latest main branch is very high.

Configuring Travis CI

Travis CI (`https://travis-ci.com/`) is a popular continuous integration service that's freely available for public GitHub projects. Integration with GitHub is very simple and it allows you to configure the platform it runs on, such as macOS, Linux, or even iOS.

Travis CI integrates tightly with GitHub, so you only need to log in to GitHub to be able to access it. We'll see how to connect our project to it.

For clarity, only the code in this chapter will be hooked up to Travis.

Travis works a bit differently from other CI tools in that it creates independent jobs by starting a new VM. This means that any artifact created for a previous stage needs to be copied somewhere else to be downloaded at the start of the next stage.

This makes things a bit unpractical sometimes, and an easy solution is to build multiple times for each individual job.

Configuring a remote system such as Travis CI can be a little frustrating sometimes, as it requires you to push a commit to be built to see if the configuration is correct. Also, it gets configured with a YAML file, which can be a bit temperamental in terms of syntax. It will take you a few attempts to get something stable, but don't worry. Once it is set up, you can change it only via a specific pull request as the configuration file is also under source control.

You can also check the requests in the Travis CI configuration to see if a `.yml` file creates a parse error.

You can check full Travis CI documentation here: `https://docs.travis-ci.com/`.

To configure Travis CI, let's start off by adding a repository from GitHub.

Adding a repo to Travis CI

To add a repo to Travis CI, we need to take the following steps:

1. The first stage is to go to the Travis CI web page and log in with your GitHub credentials.
2. Then, you'll need to grant Travis access to GitHub, by activating it.
3. Then, select which repo you want to build.

The easiest starting point is to fork the repo with the examples from this book in `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python`. Feel free to do so!

But remember to change the usernames, credentials, and registry information to match your own.

You'll need to have owner permissions for the GitHub repos and then you're ready to go!

Creating the .travis.yml file

The main element in Travis CI is the creation of the `.travis.yml` file.

Be sure to name it exactly like this (including the initial dot and the `.yml` extension) and include it in the root directory of your GitHub repo. If not, Travis CI builds won't start. Please note that, in the example repo, the file is in the **root directory** and **not** under the `Chapter04` subdirectory.

`.travis.yml` describes the build and its different steps. A build gets executed in one or more VMs. Those VMs can be configured by specifying the general operating system and the specific version. By default, they run in Ubuntu Linux 14.04 Trusty. You can find more information about available operating systems here: `https://docs.travis-ci.com/user/reference/overview/`.

Using Docker allows us to abstract most of the operating system differences, but we need to ensure the specific `docker` and `docker-compose` version that we use is correct.

We will start `.travis.yml`, ensuring that a valid `docker-compose` version (1.23.2) is present, by using the following code:

```
services:
  - docker

env:
  - DOCKER_COMPOSE_VERSION=1.23.2

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L
https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VERSIO
N}/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin
  - docker --version
  - docker-compose version
```

The `before_install` block will be executed in all our VMs. Now, to run the tests, we add a `script` block:

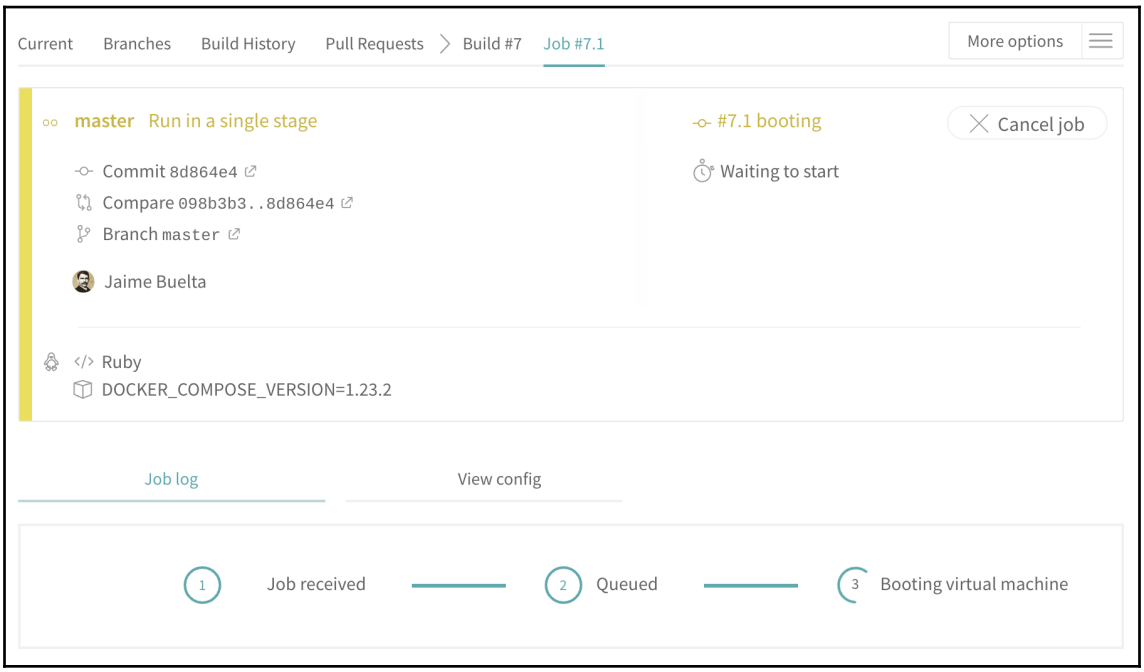
```
script:
- cd ch4
- docker-compose build db
- docker-compose build static-analysis
- docker-compose build test-postgresql
- docker-compose run test-postgresql
- docker-compose run static-analysis
```

We build all the images to use and then run the tests. Note that running the tests using the PostgreSQL database requires you to build the `db` container.

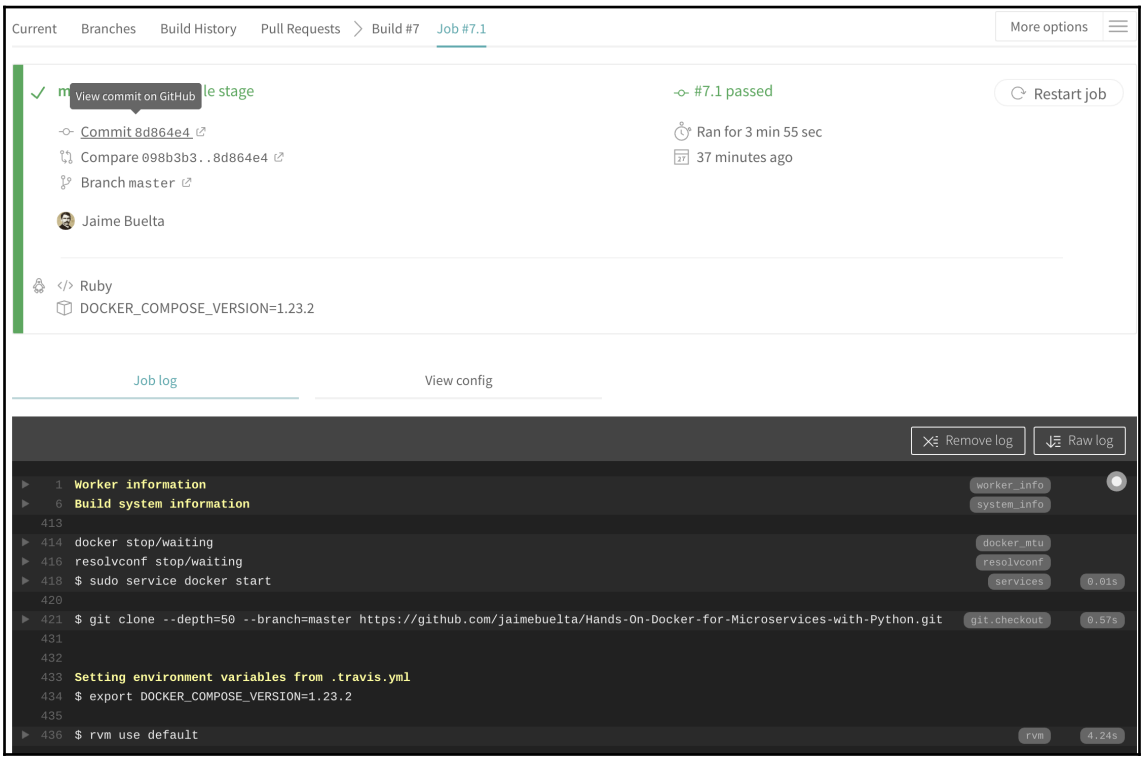
There's a small detail about the `db` container: the Travis VM doesn't allow us to open port `5432`. We removed `ports` in `docker-compose` for that. Note that this only makes PostgreSQL available externally for debugging purposes; internally, the containers can talk to each other through their internal network.

We created a `db-debug` service that's a copy of `db` but it exposes the port for local development. You can check it in the `docker-compose.yaml` file at `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter04/docker-compose.yaml`.

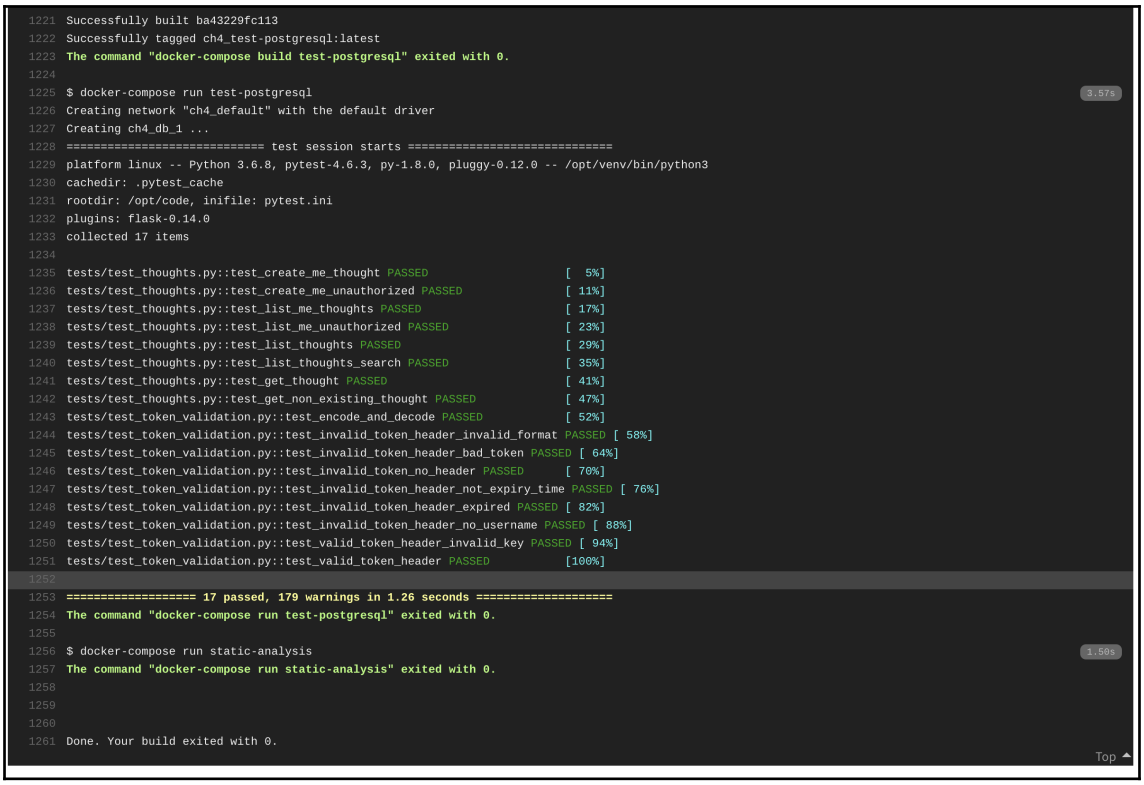
This runs all the tests. After pushing into the repo, we can see that the build starts in Travis CI:



Once it finishes, we can tell that the build was successful by the fact that it's flagged in green. The logs can then be checked for more information:



And now you can see the tests at the end of the logs:



This is useful for detecting problems and build breaks. Now, let's look at how jobs work in Travis.

Working with Travis jobs

Travis divides the whole build into a collection of stages that will run one after another. At each stage, there can be several jobs. All of the jobs in the same build will run in parallel.

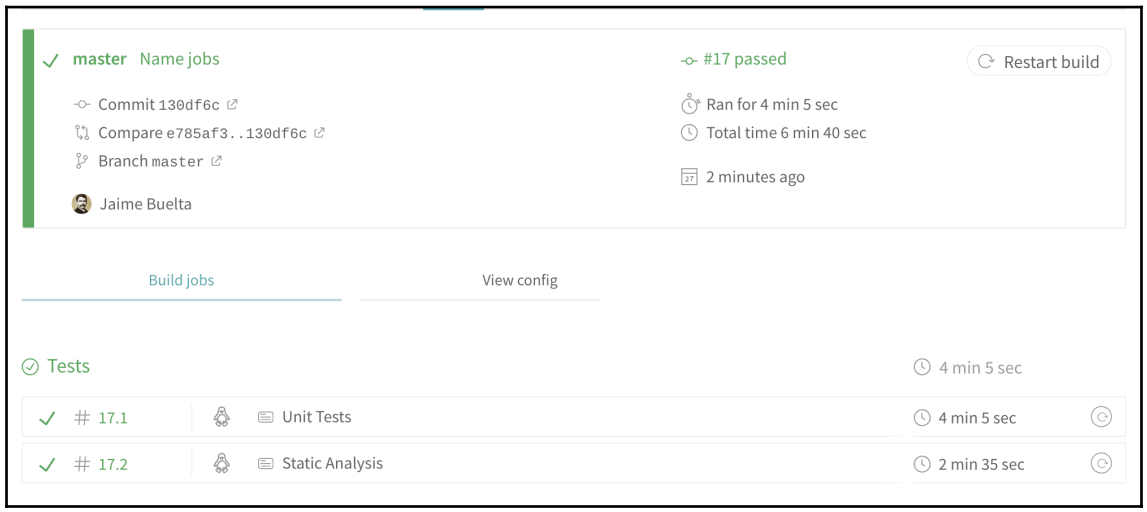
As we've seen before, we can configure tests and static analysis to run in parallel, by replacing the `script` section with a `jobs` section:

```
jobs:
  include:
    - stage: tests
      name: "Unit Tests"
```

```
    script:
    - cd ch4
    - docker-compose build db
    - docker-compose build test-postgresql
    - docker-compose run test-postgresql
  - stage: tests
    name: "Static Analysis"
    script:
    - cd ch4
    - docker-compose build static-analysis
    - docker-compose run static-analysis
```

This creates two jobs implicitly in one stage. The stage is named `tests` and the jobs are called `"Unit Tests"` and `"Static Analysis"`.

The results appear on the Travis page:



Note that, in both cases, as the jobs are independent, they need to build the required images. As the unit test job needs to build the `db` image, which takes a couple of minutes, it is slower than the static analysis one.

You can check the detailed logs on each job. Note how the environment setting and `before_install` actions get executed in all jobs.

This division can not only speed up the build quite dramatically, but it can also clarify what the problems are. At a brief glance, you can see that the breaking factor was either the unit tests or the static analysis. This removes clutter.

Sending notifications

By default, Travis CI will send an email to notify the result of a build, but only when the build is broken or when a broken build is fixed. This avoids constantly sending *success* emails and acts only when action is required. The email is only sent to the committer (and the commit author, if different) by default.

Note that there's a difference between *failed* builds and *errored* builds. The latter are failures in the job setup, which means that there's a problem in the `before_install`, `install`, or `before_script` sections, while failed builds arise because the script part returned a non-zero result. *Errored* builds are common while changing Travis configuration.

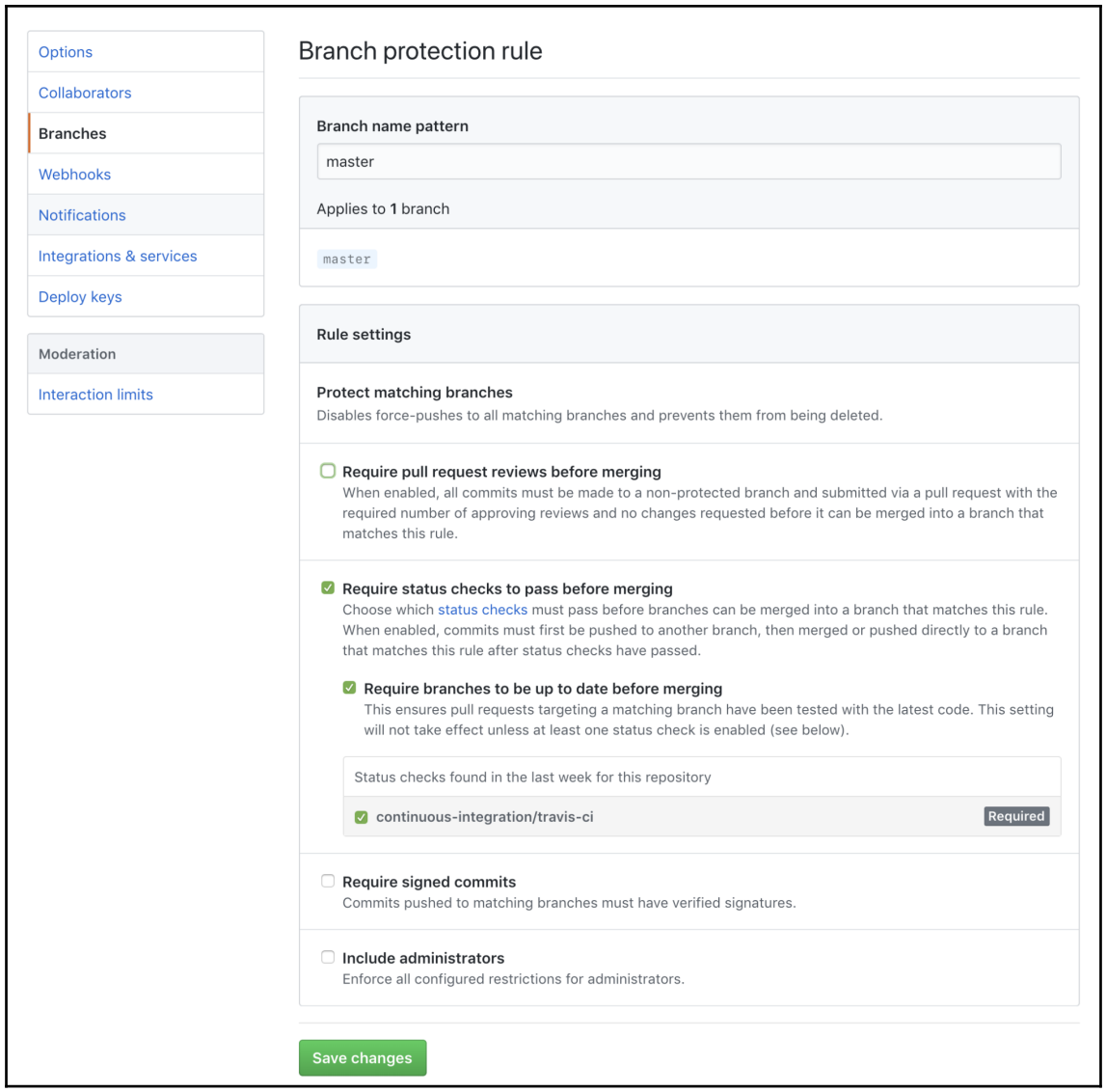
Travis allows us to configure notification emails and hook up more notification systems, including Slack, IRC, or even OpsGenie, which is capable of sending SMS messages based on on-call schedules. Check the full documentation here for more information: `https://docs.travis-ci.com/user/notifications/`.

Configuring GitHub

To take full advantage of our configured CI system, we need to ensure that we check the build before merging it into the main branch. To do so, we can configure `master` in GitHub as the main branch and add requirements before merging into it:

Be sure that the `.travis.yaml` file contains the proper credentials if you fork the repo. You'll need to update them with your own.

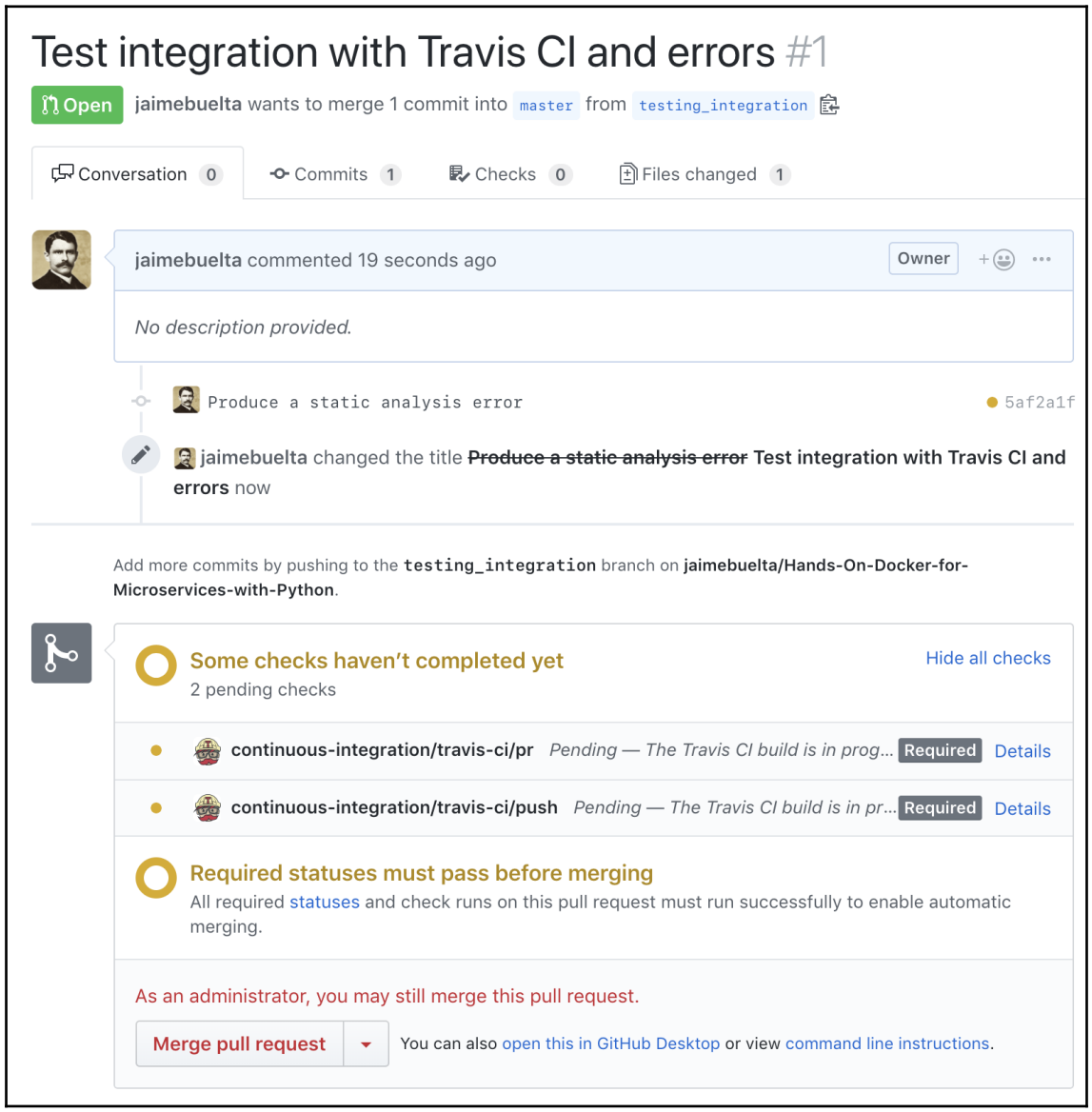
1. Go to **Settings** and **Branches** in our GitHub repo and click **Add rule**.
2. Then, we enable the **Require status checks to pass before merging** option with the status checks from `travis-ci`:



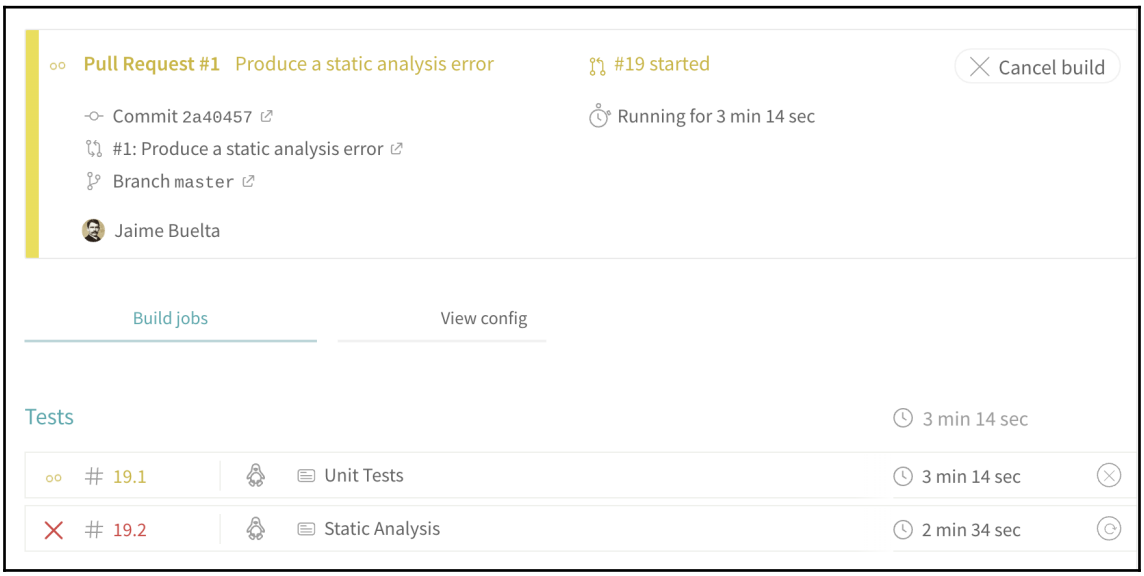
3. We also select the **Require branches to be up to date before merging** option. This ensures that there are no merges into `master` that haven't been run before.

Take a look at the other possibilities that GitHub offers. In particular, enforcing code reviews is advisable to make code to be reviewed before being merged and disseminating knowledge.

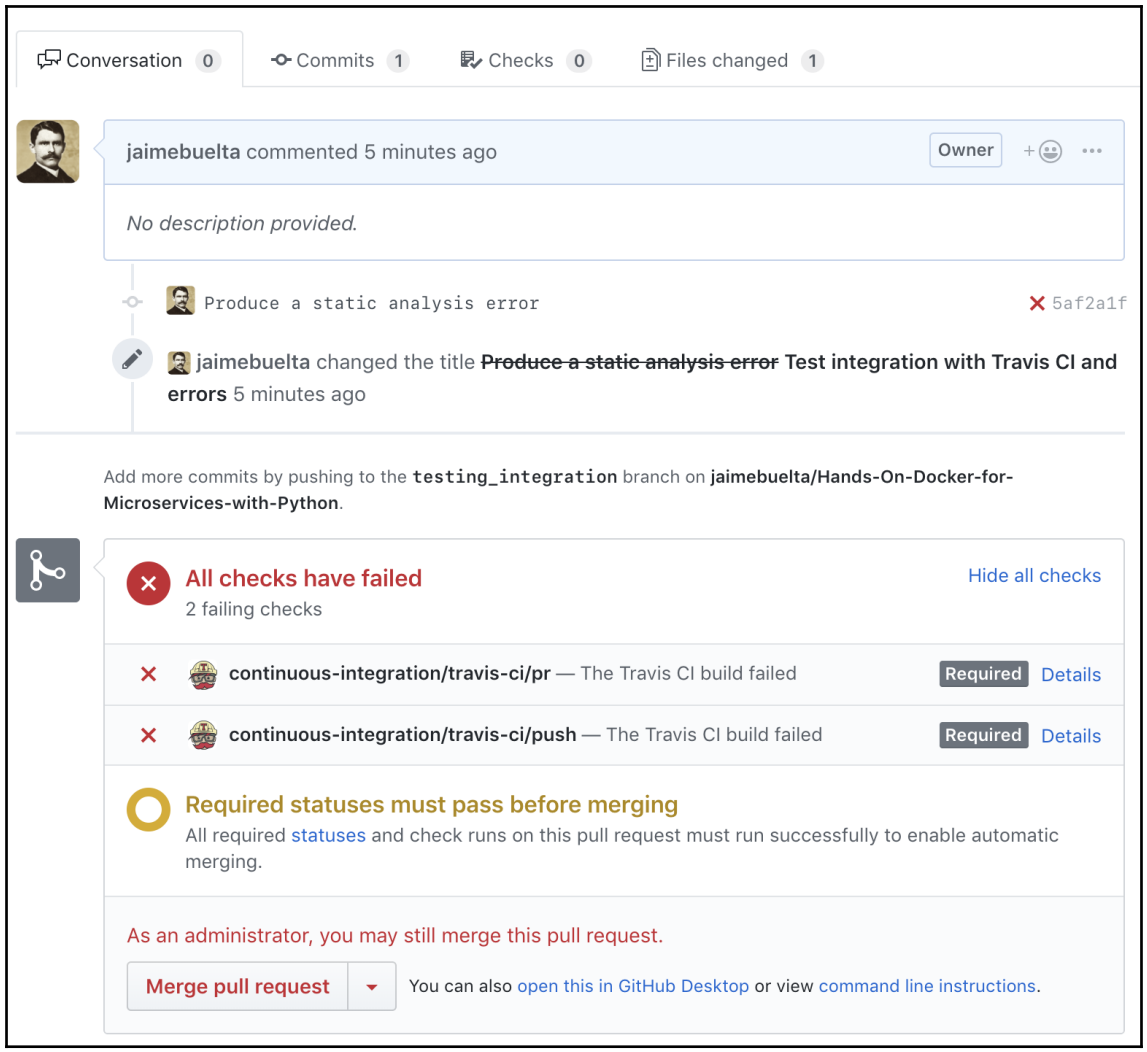
4. After creating a new branch and a new pull request designed to fail static tests, we can see how tests are added to GitHub:



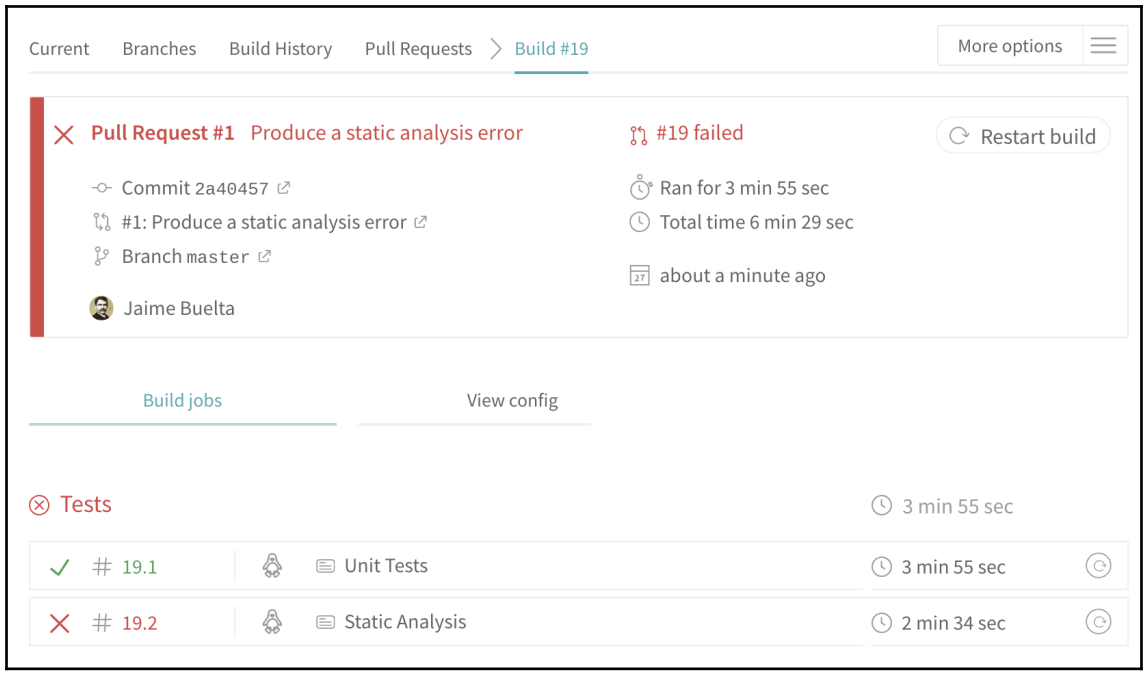
The **Details** links take you to Travis CI and the specific builds. You can also see a history of the builds:



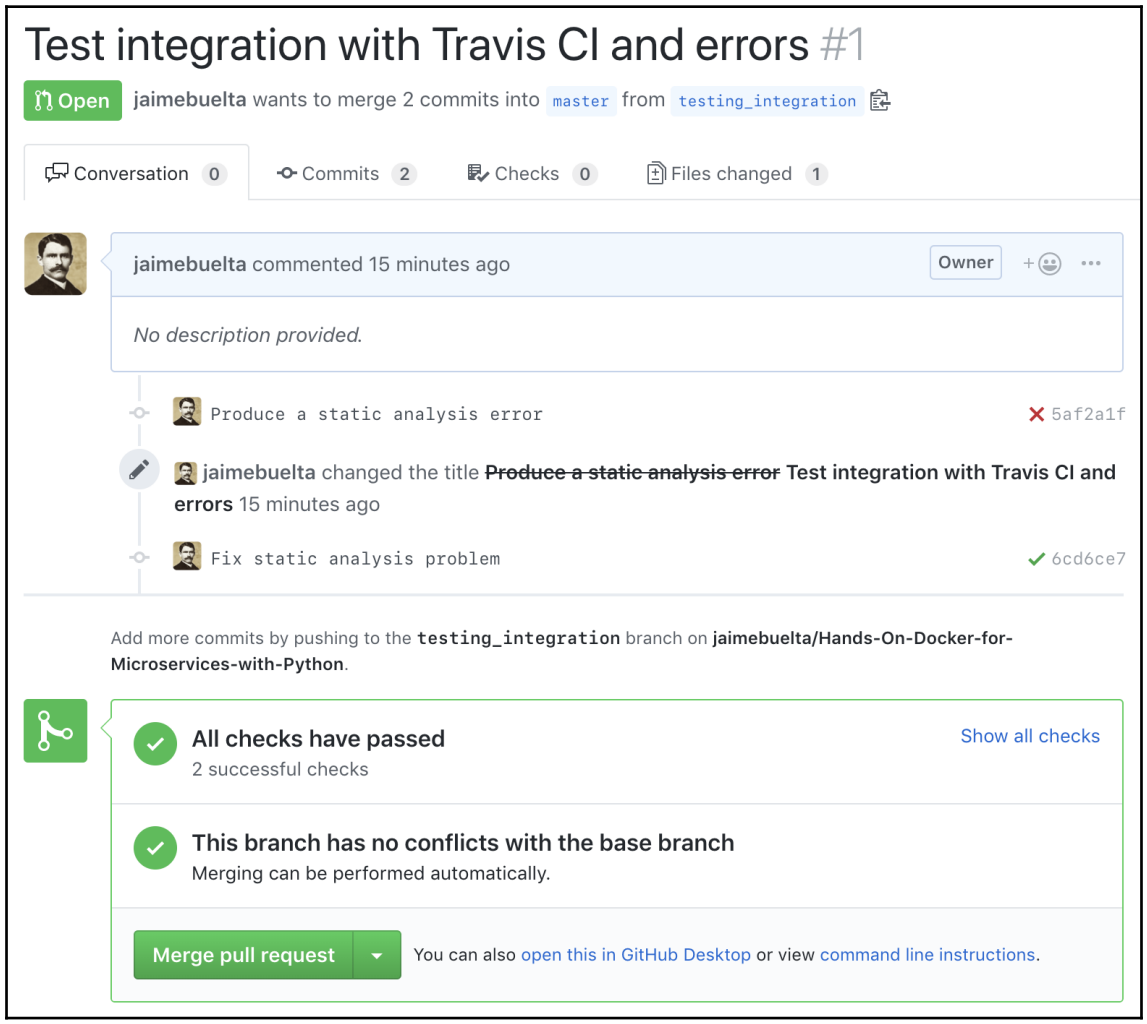
When the build is finished, GitHub won't let you merge the pull request:



The details can be found on the build page in Travis CI:



Fixing the problem and pushing the code will trigger another build. This time, it will be successful, and the pull request will be merged successfully. You can see how each commit has its own build information, whether it is correct or incorrect:



We can now merge into the master branch with confidence that the `master` branch won't break when it runs its tests.

Note that there are two builds in the pull request: one for the branch and another for the pull request. By default, Travis CI has that configuration. If you force it to always create a pull request before merging, the request will be redundant, though it can help in some cases when the branch gets pushed before creating a pull request. You can enable or disable it in the Travis project configuration.

Another interesting feature that can be configured is automatically canceling builds if a newer commit is pushed. This helps to reduce the total number of builds in the system.

Build result can also be checked in the **Commits** view in GitHub.

Pushing Docker images from Travis CI

After our build creates a Docker image, we need to be able to share it with the rest of the team or deploy it. We will use the Docker registry in Docker Hub, as explained in the previous chapter, to push the images.

Let's start by setting the secret variables.

Setting the secret variables

To be able to push to the Docker repo, we first need to configure a password to log in to the Docker registry. This needs to be done through the secrets configuration in Travis CI, to avoid committing sensible information in the GitHub repo:

It's worth repeating: **do not commit secrets in your GitHub repo**. These techniques can be used for any other required secret.

1. Install the `travis` command line using `gem`. This assumes that you have `gem` installed on your system (Ruby 1.93 or later). If you don't, check the installation instructions (`https://github.com/travis-ci/travis.rb#installation`):

```
$ gem install travis
```

2. Log in to Travis:

```
travis login --pro
```

3. Create a secure variable with the Docker Hub username:

```
$ travis encrypt --com DOCKER_USERNAME="<your user name>"
```

4. You'll see output similar to the following:

```
secure: ".... encrypted data ...."
```

5. Then, you need to add the encrypted data to the environment variables, as follows:

```
env:
  global:
    - DOCKER_COMPOSE_VERSION=1.23.2
    - secure: ".... encrypted data ...."
```

6. Now, note the new `global` section and repeat step 3 with the Docker Hub password:

```
$ travis encrypt --com DOCKER_PASSWORD="<password>"
```

7. Add another secure variable, after the first one:

```
env:
  global:
    - DOCKER_COMPOSE_VERSION=1.23.2
    - secure: ".... encrypted data ...."
    - secure: ".... encrypted data ...."
```

This operation creates two environment variables, available during the builds. Do not worry—they will be not shown in the logs:

```
Setting environment variables from .travis.yml
$ export DOCKER_COMPOSE_VERSION=1.23.2
$ export DOCKER_PASSWORD=[secure]
$ export DOCKER_USERNAME=[secure]
```

We can now add the proper login command in the `before_install` section so that Docker service can connect and push images:

```
before_install:
  ...
  - echo "Login into Docker Hub"
  - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --
password-stdin
```

The next stage is to build and tag the resulting image.

Tagging and pushing builds

The following code will add a new stage that will build, tag, and finally push the result to the Docker registry:

```
jobs:
  include:
    ...
    - stage: push
      script:
      - cd Chapter04
      - docker-compose build server
      - docker tag thoughts_server:latest <registry>/thoughts-
backend:$TRAVIS_BRANCH
```

This first part builds the final image for the server and tags it with the name of the branch. To deploy it, we will add a `deploy` section:

```
- stage: push
  script:
  ...
  - docker tag thoughts_server:latest <registry>/thoughts-
backend:$TRAVIS_BRANCH
  deploy:
  - provider: script
    script: docker push <registry>/thoughts-backend:$TRAVIS_BRANCH
    on:
      branch: master
```

The `deploy` section will execute a `script` command when the branch is `master`. Now, our build will also generate a final image and push it. This will ensure our registry gets the latest version in our main branch.

We can add more `deploy` conditions to push the tag; for example, if we create a new Git tag, we can push the resulting image with the proper tag.

Remember that tags, as discussed in the previous chapter, are a way to mark an image as significant. Normally, this will mean it's ready for some to be used outside automatic tests, for example, in deployment.

We can add the tags to the `deploy` section:

```
    deploy:
    - provider: script
      script: docker push <registry>/thoughts-backend:$TRAVIS_BRANCH
      on:
```

```
      branch: master
  - provider: script
    script: docker push <registry>/thoughts-backend:$TRAVIS_TAG
    on:
      tags: True
```

Note that here we push whether the branch is the master or there's a defined tag, as both conditions won't be matched.

You can check the full deployment documentation here: `https://docs.travis-ci.com/user/deployment`. We've covered the `script` provider, which is a way of creating our own commands, but offers support for providers such as Heroku, PyPI (in the case of creating a Python package), and AWS S3.

Tagging and pushing every commit

It is possible to push every single built image to the registry, identified by its Git SHA. This can be useful when work in progress can be shared for demo purposes, tests, and so on.

To do so, we need to create an environment variable with the Git SHA in the `before_install` section:

```
before_install:
  ...
  - export GIT_SHA=`git rev-parse --short HEAD`
  - echo "Building commit $GIT_SHA"
```

The `push` section then adds the tag and push of the image:

```
- stage: push
  script:
  - cd Chapter04
  - docker-compose build server
  - docker tag thoughts_server:latest <registry>/thoughts-backend:$GIT_SHA
  - docker push <registry>/thoughts-backend:$GIT_SHA
  - docker tag thoughts_server:latest <registry>/thoughts-
backend:$TRAVIS_BRANCH
```

As this action happens before the `deploy` part, it will be produced on every build that reaches this section.

This method will produce a lot of tags. Depending on how your registry manages them, that may be costly. Be sure that it is a sensible thing to do.

Keep in mind that this same approach can be used for other conditional pushes.

Please note that the registry needs to be adapted to your own registry details. If you clone the example repo, the later will need to be changed.

Summary

In this chapter, we presented continuous integration practices and explored how Docker helps to implement them. We also looked at how to design a pipeline that ensures that our code always follows high standards and detects deviations as soon as possible. Using Git branches and pull requests in GitHub plays along with this, as we can determine when the code is ready to be merged into the main branch and deployed.

We then introduced Travis CI as a great tool to work with alongside GitHub to achieve continuous integration, and discussed its features. We learned how to create a pipeline in Travis CI, from the creation of the `.travis.yml` file, how to configure jobs, how to make the build push a validated Docker image to our Docker registry, and how to be notified.

We described how to speed up running sections in parallel, as well as how to set values as secrets. We also configured GitHub to ensure that the Travis CI pipeline has run successfully before merging new code into our main branch.

In the next chapter, we will learn about basic Kubernetes operations and concepts.

Questions

1. Does increasing the number of deployments reduce their quality?
2. Describe what a pipeline is.
3. How do we know if our main branch can be deployed?
4. What is the main configuration source for Travis CI?
5. When will Travis CI send a notification email by default?
6. How can we avoid merging a broken branch into our main branch?
7. Why should we avoid storing secrets in a Git repo?

Further reading

To learn more about continuous integration and other tools, you can check out the book *Hands-On Continuous Integration and Delivery* (`https://www.packtpub.com/eu/virtualization-and-cloud/hands-continuous-integration-and-delivery`), which covers not only Travis CI but other tools such as Jenkins and CircleCI. If you want to dig deeper into GitHub and all its possibilities, including how to effectively collaborate and the different workflows it enables, find out more in *GitHub Essentials* (`https://www.packtpub.com/eu/web-development/github-essentials-second-edition`).

3
Section 3:Working with Multiple Services – Operating the System through Kubernetes

In the last section, we covered how to develop and containerize a single microservice, and this section introduces the orchestration concept, making multiple services work in unison. In this section, Kubernetes, as an orchestrator of Docker containers, is explained in depth, along with practices to maximize its usage and how to deploy it for real-world operations.

The first chapter of this section introduces Kubernetes and explains the basic concepts behind this tool that will be used throughout the section. Kubernetes has its own specific nomenclature that can initially be a bit overwhelming, so don't be afraid to come back to this chapter when something is not clear. It also covers how to install and operate a local cluster.

The second chapter of this section shows how to install the developed microservices in a local Kubernetes cluster, using the concepts introduced in the previous chapter in a concrete operation. It configures a full cluster with services running and cooperating, while also demonstrating how to develop in this environment.

The third chapter of this section deals with a real-life operation: how to create a cloud cluster using commercial cloud services (we use AWS services in this book) that is aimed at providing a service to external customers in the open internet. It also covers how to secure the service under proper HTTPS endpoints, using private Docker registries, and advanced topics such as automatic scaling of the cluster and practices to ensure the smooth running of containers.

The fourth chapter of this section introduces the concept of GitOps, which entails using a Git repository to control the cluster infrastructure, keeping any infrastructure change under source control, and allowing the use of common elements of Git, such as pull request, to control and verify the fact that infrastructure changes are correct.

The fifth chapter of this section describes the software life cycle inside a single service and how adding a new feature works, since the feature is defined until it is running live in the existing Kubernetes cluster. This chapter shows practices for testing and verifying new features to introduce them into the live system with confidence and in an efficient manner.

This section comprises the following chapters:

- `Chapter 5`, *Using Kubernetes to Coordinate Microservices*
- `Chapter 6`, *Local Development with Kubernetes*
- `Chapter 7`, *Configuring and Securing the Production System*
- `Chapter 8`, *Using GitOps Principles*
- `Chapter 9`, *Managing Workflows*

5
Using Kubernetes to Coordinate Microservices

In this chapter, we will talk about the basic concepts behind Kubernetes, a tool that allows you to manage multiple containers and coordinate them, thereby making the microservices that have been deployed on each container work in unison.

This chapter will cover what a container orchestrator is and specific Kubernetes nomenclature, such as the differences between a pod, a service, a deployment, and so on. We will also learn how to analyze a running cluster and perform other common operations so that you can apply them to our microservices example.

In this chapter, we will cover the following topics:

- Defining the Kubernetes orchestrator
- Understanding the different Kubernetes elements
- Performing basic operations with kubectl
- Troubleshooting a running cluster

By the end of this chapter, you'll know about the basic elements of Kubernetes and will be able to perform basic operations. You'll also learn about basic troubleshooting skills so that you can detect possible issues.

Technical requirements

If you're working with macOS or Windows, the default Docker desktop installation can start a local Kubernetes cluster. Just ensure that this is enabled in Kubernetes' preferences:

For Linux, the easiest way to install Kubernetes locally is to use k3s (`https://k3s.io/`).

k3s is a nod to Kubernetes (that is, k8s) but is a simplified version of it.

k3s is a minimalistic installation of Kubernetes that you can use to run a cluster contained in a single binary. Check out the installation page (`https://github.com/rancher/k3s/blob/master/README.md`) if you wish to download and run it.

To be able to use the Docker version that's running inside the k3s cluster, we need to use the following code:

```
$ # Install k3s
$ curl -sfL https://get.k3s.io | sh -
$ # Restart k3s in docker mode
$ sudo systemctl edit --full k3s.service
# Replace `ExecStart=/usr/local/bin/k3s` with `ExecStart=/usr/local/bin/k3s
server --docker`
$ sudo systemctl daemon-reload
$ sudo systemctl restart k3s
$ sudo systemctl enable k3s
$ # Allow access outside of root to KUBECTL config
$ sudo chmod 644 /etc/rancher/k3s/k3s.yaml
$ # Add your user to the docker group, to be able to run docker commands
$ # You may need to log out and log in again for the group to take effect
$ sudo usermod -a -G docker $USER
```

Ensure that you install `kubectl` (k3s installs a separate version of it by default). The steps to install `kubectl` can be found at `https://kubernetes.io/docs/tasks/tools/install-kubectl/`. The `kubectl` command controls Kubernetes operations.

Check the instructions on the aforementioned page to add Bash completion, which will allow us to hit *Tab* to complete some commands.

If everything has been correctly installed, you should be able to check the running pods with the following commands:

```
$ kubectl get pods --all-namespaces
NAMESPACE NAME                                     READY STATUS
RESTARTS AGE
docker    compose-89fb656cf-cw7bb                  1/1   Running 0
1m
docker    compose-api-64d7d9c945-p98r2             1/1   Running 0
1m
kube-system etcd-docker-for-desktop                1/1   Running 0
260d
kube-system kube-apiserver-docker-for-desktop      1/1   Running 0
2m
kube-system kube-controller-manager-docker-for-desktop 1/1   Running 0
2m
kube-system kube-dns-86f4d74b45-cgpsj              3/3   Running 1
260d
kube-system kube-proxy-rm82n                       1/1   Running 0
2m
```

```
kube-system kube-scheduler-docker-for-desktop          1/1   Running 0
2m
kube-system kubernetes-dashboard-7b9c7bc8c9-hzpkj      1/1   Running 1
260d
```

Note the different namespaces. They are all default ones that were created by Kubernetes itself.

Go to the following page to install the Ingress controller: `https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md`. In Docker desktop, you'll need to run these two commands:

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/st
atic/mandatory.yaml
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/st
atic/provider/cloud-generic.yaml
```

This will create an `ingress-nginx` namespace with a controller pod. Kubernetes will use that pod to set up the Ingress configuration.

Now, let's take a look at the advantages of using Kubernetes.

Defining the Kubernetes orchestrator

Kubernetes is a popular container orchestration tool. It allows us to manage and deploy multiple containers that interact with each other in a coordinated way. Since each microservice lives in an individual container, as we mentioned in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, they can work in unison.

For a more in-depth introduction to Kubernetes, you can check out the following comic, which was released by Scott McCloud: `https://cloud.google.com/kubernetes-engine/kubernetes-comic/`.

Kubernetes is aimed at production systems. It was designed to be able to control big deployments and to abstract most of the infrastructure's details. Every element in a Kubernetes cluster is configured programmatically, and Kubernetes itself manages where to deploy clusters based on the capacity that's available.

Kubernetes can be configured completely using configuration files. This makes it possible to replicate clusters, for example, in the event of a full disaster that brings down all the physical servers. You can even do this with different hardware, where traditional deployments could be extremely difficult.

This example assumes that the data is stored and retrievable; for example, in a backup device. Obviously, this may be difficult—disaster recovery always is. However, it simplifies a lot of the steps that are required if you wish to replicate a cluster.

Given that Kubernetes works with containers and makes it very easy to install them, there's a big ecosystem of containers ready to add functionality to Kubernetes itself. The best example is probably the Kubernetes dashboard (`https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/`), a UI that displays an overview of Kubernetes' operations. It isn't installed by default, but it can be installed in the same way that you install your services. Other examples for such use cases include monitoring and logging. This makes Kubernetes very extendable.

Comparing Kubernetes with Docker Swarm

Kubernetes is not the only orchestrator that's available. As we mentioned in `Chapter 3`, *Build, Run, and Test Your Service Using Docker*, there's `docker-compose`. Docker Compose can also orchestrate different containers and coordinate them but does so without dealing with multiple servers.

Docker has a native extension of `docker-compose` called Docker Swarm. This allows us to use a cluster of machines to run `docker-compose` while reusing the same YAML files, but with a few added details to describe how you want them to run.

You can learn more about Docker Swarm in the official documentation (`https://docs.docker.com/engine/swarm/`).

Docker Swarm is easier to set up than Kubernetes, assuming that you have to manage the servers. As you extend the functionality of Docker Compose, you'll find that it has a low learning curve.

On the other hand, Kubernetes is way more powerful and customizable. It has a bigger community and a higher pace of innovation. It's also better at handling issues. The biggest problem is setting up a cluster, but as we'll see in `Chapter 7`, *Configuring and Securing the Production System*, nowadays, there are easy commercial deployments we can use to create a cluster in a few minutes, which lowers the barrier of entry for Kubernetes.

This makes Kubernetes (arguably) a better solution when you're dealing with migrating from the old system and when looking toward the future. For small deployments, or if you need to deploy and manage your own servers, Docker Swarm can be an interesting alternative.

To help you move on from using a `docker-compose.yaml` file to using the equivalent Kubernetes YAML files, you can use `kompose` (`https://github.com/kubernetes/kompose`). It may be useful to quickly start a Kubernetes cluster and translate the services described in the `docker-compose.yaml` file into their equivalent Kubernetes elements, but there are always differences between both systems that may need to be tweaked.

Let's start by describing the specific elements and nomenclature of Kubernetes.

Understanding the different Kubernetes elements

Kubernetes has its own nomenclature for different elements. We will be using the nomenclature often in this book, and the Kubernetes documentation uses them as well. Learning about the differences between them is important since some of them can be subtle.

Nodes

The main infrastructure elements of Kubernetes are known as **nodes**. A Kubernetes cluster is composed of one or more nodes, which are the physical machines (or virtual machines) that support the abstraction of the rest of the elements.

Each node needs to be able to communicate with the others, and they all run in a *container runtime*—typically Docker—but they can use other systems, such as `rktlet` (`https://github.com/kubernetes-incubator/rktlet`).

The nodes create a network between them that routes all the requests that have been addressed to the cluster so that any request that's sent to any node in the cluster will be answered adequately. Kubernetes will handle what deployable goes to what node, even recovering nodes if they go down or moving them around from one node to another if there are resources issues.

Nodes don't necessarily need to be identical, and some degree of control is needed when it comes to deploying specific elements in specific nodes, but for simplicity, they normally are identical.

While nodes are the backbone that supports the cluster, Kubernetes helps in abstracting away from specific nodes by defining the desired outcome and letting Kubernetes do the heavy lifting of deciding what goes where and being sure that the internal network channels' requests are sent to the proper services.

Kubernetes Control Plane

The Kubernetes Control Plane is where all the processes that Kubernetes uses to properly configure a collection of servers as nodes in a Kubernetes cluster are kept. Servers allow nodes to connect to each other, allow us to monitor their current status, and allows us to make whatever changes are necessary in terms of deployment, scale, and so on.

The node that's responsible for registering and making these changes is called the master node. There can be more than one master node.

All of this control normally runs smoothly behind the scenes. Its network is separated from the rest, meaning that a problem at this level won't affect the current operation of the cluster, other than us not being able to make changes.

Kubernetes Objects

Kubernetes Objects are abstractions that represent the state of the service that's deployed in the cluster. Mainly, they deal with running containers and routings for those containers, as well as persistent storage.

Let's take a look at the different elements, from smallest to biggest. This list is not exhaustive; check out the Kubernetes documentation for more details:

- **Container**: A single Docker container. These are the building blocks of Kubernetes, but they're never present on their own.
- **Pod:** A basic unit that can be deployed in Kubernetes. A pod is a collection of one or more containers that work as a unit, normally from different images. Normally, a pod has a single container, but sometimes it may be useful to have more. All of the containers in the same pod share the same IP address (the pod IP), meaning that a container that accesses a port in `localhost` may be accessing a different container instead. This is actually the recommended way of communicating with them.

This will all be a bit strange to you at first, but normally, multi-container pods will have a main container and something else that performs auxiliary tasks, such as exporting metrics.

- **ConfigMap**: This defines a set of key-value pairs that can be injected into pods, typically as environment variables or files. This allows us to share configurations between different defined pods, for example, to make all the containers log debug information. Note that pods can have their own configuration, but ConfigMaps are a convenient way to share the same values so that they are available to different pods.
- **Volume**: The files that are inside a container are ephemeral and will be lost if the container stops its execution. A volume is a form of persistent storage that can be used to maintain data information between starts and to share information between containers in a pod.

As a general principle, try to have as few volumes as possible. Most of your applications should be stateless anyway, and any variable data should be stored in a database. If containers in the same pod need to communicate, it is better to do so through HTTP requests. Remember that any immutable data, such as static files, can be stored inside the container image.

- **Deployment**: This is a grouping of one or more identical pods. The definition of the deployment will state the desired number and Kubernetes will work to get to this, according to whatever strategy is defined. The pods in a single deployment can be deployed to different nodes, and normally will be. If any of the pods are deleted, finished, or have any kind of problem, the deployment will start another until the defined number is reached.
- **Job**: A job creates one or more pods that are expected to finish. While a deployment will assume that any pod that's finishing is a problem and will start another, jobs will retry until the proper number of successes is met. The finished pods are not deleted, which means we can check their logs. Jobs are one-off executions. There are also **Cron Jobs**, which will run at specific times.
- **Service**. Since pods are created and recreated and have different IPs, to allow services to access them, a service needs to define the name that other elements can use to discover it. In other words, it routes requests to the proper pods. Normally, a service and a deployment will be related, with the service making the deployment accessible and round-robin between all the defined pods. A service can also be used to create an internal name for an external service.

Services in Kubernetes solve an old problem in distributed systems, that is, *service discovery*. This problem occurs when nodes in a cluster need to know where a service lives, even if the nodes change; that is, when we add a node or remove it without changing the configuration settings of all the nodes.

Kubernetes will do this automatically if you create a service.

- **Ingress**: While a service is internal, an Ingress is external. It routes any external requests to the appropriate service so that they can be served. You can define different Ingresses by host name, which ensures that the cluster is routed to different services by the target host of the request, or a single Ingress is hosted in terms of its path. Internally, an Ingress is implemented as a container that implements the Ingress controller, which is `nginx` by default.

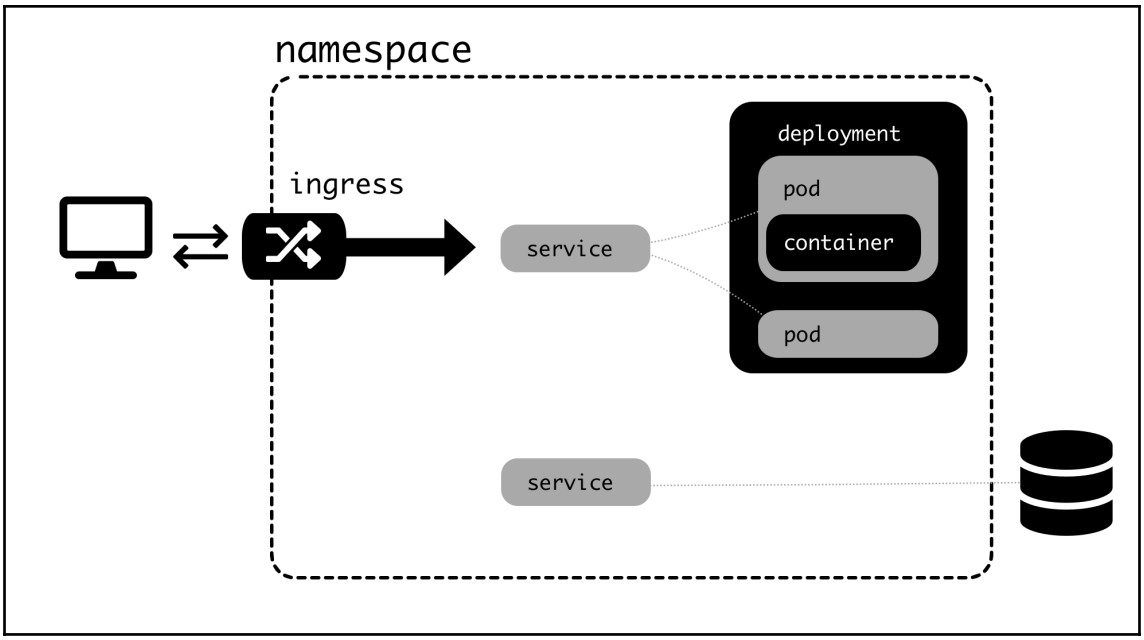
Depending on your Kubernetes installation, you may need to install the default controller. To install the default controller, follow the instructions at `https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md`.

- **Namespace**: This is the definition of a virtual cluster. You can define more than one namespace in the same physical Kubernetes cluster. Every name that's defined under a namespace needs to be unique, but another namespace could use the same definition. Objects in different namespaces can't communicate with each other internally, but they can do so externally.

Generating different namespaces with very similar definitions can be useful if you wish to create different environments for purposes, such as testing, development, or demo concepts. The main advantage of Kubernetes is that you can replicate a whole system and take advantage of this to create similar environments with small changes in terms of details, such as a new version of an environment.

Objects can be found in `.yaml` files, which can be loaded into the system. A single `.yaml` file can define multiple objects, for example, a deployment that defines pods that contain containers.

The following diagram summarizes the different objects that are available:



Jobs and volumes are not present, but two services are available: one that points toward a deployment and another that points toward an external service. The external service is aimed at internal elements and isn't exposed externally.

Performing basic operations with kubectl

By using `kubectl`, we can perform operations against all the different elements. We've already had a sneak peek at `get` to get an idea of what elements are available.

For more information and a quick overview of the most common operations that are available within `kubectl`, check out the `kubectl` cheat sheet at `https://kubernetes.io/docs/reference/kubectl/cheatsheet/`.

We can use `kubectl` to `create` a new element. For example, to create and list namespaces, we can use the following code:

```
$ kubectl create namespace example
namespace/example created
$ kubectl get namespaces
NAME          STATUS AGE
default       Active 260d
docker        Active 260d
example       Active 9s
kube-public   Active 260d
kube-system   Active 260d
```

We can create various elements, some of which we'll look at throughout this book.

Defining an element

A namespace is a special case as it doesn't require any configuration. To create a new element, a YAML file needs to be created that describes that element. For example, we can create a new pod using the official NGINX image in Docker Hub:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: example
spec:
  containers:
    - name: nginx
      image: library/nginx:latest
```

As a minimum, an element should contain the following:

- The API version of the element.
- The element's type.
- A name for the element, as well as a namespace for it.
- A `spec` section that includes configuration details. For a pod, we need to add the necessary containers.

YAML files can be a bit temperamental sometimes, especially when it comes to indentation and syntax. You can use a tool such as Kubeval (`https://kubeval.instrumenta.dev/`) to check that the file is correct and that you're following Kubernetes good practices before using a file.

We will save this file as `example_pod.yml`. We'll create it by using the `apply` command and monitor that it's running with the following commands:

```
$ kubectl apply -f example_pod.yml
pod/nginx created
$ kubectl get pods -n example
NAME  READY STATUS            RESTARTS AGE
nginx 0/1   ContainerCreating 0        2s
$ kubectl get pods -n example
NAME  READY STATUS  RESTARTS AGE
nginx 1/1   Running 0        51s
```

Note the usage of the `-n` parameter to determine the namespace.

We can now `exec` into the container and run commands inside it. For example, to check that the NGINX server is running and serving files, we can use the following code:

```
$ kubectl exec -it nginx -n example /bin/bash
root@nginx:/# apt-get update
...
root@nginx:/# apt-get install -y curl
...
root@nginx:/# curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

```
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

The pod can be changed in two ways. The first way is to manually run `edit`, which opens your predefined Terminal editor so that you can edit the file:

```
$ kubectl edit pod nginx -n example
```

You'll see the pod with all its default parameters. This way of changing a pod is useful for small tests, but in general, it's better to change the original YAML file so that you can keep track of the changes that occur. For example, we can change NGINX so that we're using a previous version of it:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: example
spec:
  containers:
    - name: nginx
      image: library/nginx:1.16
```

Then, we can `apply` these changes once more, which will restart the pod:

```
$ kubectl apply -f example_pod.yml
pod/nginx configured
$ kubectl get pods -n example
NAME  READY STATUS  RESTARTS AGE
nginx 1/1   Running 1        36s
```

Getting more information

The `get` command accepts more configuration. You can retrieve more information by using the `wide` output option:

```
$ kubectl get pods -n example -o wide
NAME  READY STATUS  RESTARTS AGE IP         NODE
nginx 1/1   Running 1        30m 10.1.0.11 docker-for-desktop
```

If you make a change and are interested in the changes that this produces, you can use the `-w` argument to watch any change. For example, the following code shows the restart results of a pod. This restart has been produced due to a change being made to the image of the container:

```
$ kubectl get pods -n example -w
NAME  READY STATUS  RESTARTS AGE
nginx 1/1   Running 2        32m
nginx 1/1   Running 3        32m
```

If you need more information about a particular element, you can `describe` it:

```
$ kubectl describe pod nginx -n example
Name: nginx
Namespace: example
Node: docker-for-desktop/192.168.65.3
Start Time: Sun, 23 Jun 2019 20:56:23 +0100
Labels: <none>
Annotations: ...
Status: Running
IP: 10.1.0.11
...
Events:
  Type Reason Age From Message
  ---- ------ ---- ---- -------
  Normal Scheduled 40m default-scheduler Successfully assigned nginx to
docker-for-desktop
  ...
  Normal Created 4m43s (x5 over 40m) kubelet, docker-for-desktop Created
container
  Normal Started 4m43s (x5 over 40m) kubelet, docker-for-desktop Started
container
```

This returns a lot of information. The most useful information is normally about events, which will return information about the life cycle of the element.

Removing an element

The `delete` command removes an element and everything under it:

```
$ kubectl delete namespace example
namespace "example" deleted
$ kubectl get pods -n example
No resources found.
```

Be aware that, sometimes, deleting an element will cause it to be recreated. This is quite common when pods are created through deployments since the deployment will work to get the number of pods to the configured number.

Troubleshooting a running cluster

The main tools that we can use to troubleshoot issues in Kubernetes are the `get` and `describe` commands.

In my experience, the most common problem with running Kubernetes is that, sometimes, certain pods don't start. The steps for troubleshooting are as follows:

1. Is the container image correct? A problem with downloading the image will show `ErrImagePull`. This could be caused if the image can't be downloaded from the registry due to an authentication problem.
2. A status of `CrashLoopBackOff` means that the process for the container has been interrupted. The pod will try to restart over and over. This is normally caused by an underlying issue with the container. Check that the configuration is correct. You can check the `stdout` logs of a container by using the following command:

   ```
   $ kubectl logs <pod> -n <namespace> -c <container>
   ```

   Ensure that the container is runnable. Try to run it manually using the following command:

   ```
   $ docker run <image>
   ```

3. A pod is not exposed externally. This is typically due to a problem in the service and/or Ingress that exposes them. You can detect whether a pod is responsive inside the cluster by using `exec` to get into another container and then try to access the service and the internal IP of the pod, normally using `curl`.

As we saw previously, `curl` is not installed in containers by default because, normally, they only install a minimal set of tools. Don't worry—you can install it using whatever package manager your operating system uses, with the advantage that, once the container is recycled (which will happen soon enough in a normal Kubernetes operation), it won't be using up any space! For the same reason, you may need to install it each time you need to debug a problem.

Remember the chain we discussed for Ingress, services, deployments, and pods and work from the inside out to find out where the misconfiguration is.

While troubleshooting, remember that pods and containers can be accessed through `exec` commands, which will allow us to check running processes, files, and much more. This is similar to accessing the Terminal of a physical server. You can do this using the following code:

```
$ kubectl exec -it <pod> -n <namespace> /bin/sh
```

Be careful as the nature of Kubernetes clusters may require you to check a specific container in a pod if there is more than one container running in the same pod.

Summary

In this chapter, we looked at the basic concepts of Kubernetes and how it's useful to manage and coordinate multiple containers that contain our microservices.

First, we introduced what Kubernetes is and some of its high-level advantages. Then, we described the different elements that define a cluster in the Kubernetes nomenclature. This included both the physical aspects, where the nodes are the main defining elements, as the abstract aspects, such as the pods, deployments, services, and Ingress, which are the building blocks we need in order to generate a working cluster.

We described `kubectl` and the common operations we can use to define elements and retrieve information through YAML files. We also described some of the common problems that can arise when handling a Kubernetes cluster.

In the next chapter, we will define the different options we can use in YAML files in order to generate clusters and learn how to generate a Kubernetes cluster for our microservices example.

Questions

1. What is a container orchestrator?
2. In Kubernetes, what is a node?
3. What is the difference between a pod and a container?
4. What is the difference between a job and a pod?
5. When should we add an Ingress?
6. What is a namespace?
7. How can we define a Kubernetes element in a file?
8. What is the difference between the `get` and `describe` commands of `kubectl`?
9. What does a `CrashLoopBackOff` error indicate?

Further reading

You can learn more about Kubernetes by reading *Getting Started with Kubernetes – Third Edition* (`https://www.packtpub.com/eu/virtualization-and-cloud/getting-started-kubernetes-third-edition`) and *The Complete Kubernetes Guide* (`https://www.packtpub.com/eu/virtualization-and-cloud/complete-kubernetes-guide`).

6
Local Development with Kubernetes

In this chapter, you'll learn how to define a cluster, deploying all the interacting microservices, and how to work locally for development purposes. We will build on the concepts introduced in the previous chapter and we will describe how to configure the whole system in Kubernetes in practical terms, deploying multiple microservices, and how to make it work as a whole on your own local computer.

Here, we will introduce the other two microservices: the Frontend and the Users Backend. They were discussed in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, in the *Strategic planning to break the monolith* section. We will see in this chapter how they need to be configured to work in Kubernetes. This is in addition to the Thoughts Backend introduced in `Chapter 2`, *Creating a REST Service with Python;* `Chapter 3`, *Build, Run, and Test Your Service Using Docker*, and `Chapter 4`, *Creating a Pipeline and Workflow*. We will discuss how to configure the three of them properly and add some other options to ensure their smooth operation once they're deployed in a production environment.

The following topics will be covered in this chapter:

- Implementing multiple services
- Configuring the services
- Deploying the full system locally

By the end of the chapter, you will have a working local Kubernetes system with the three microservices deployed and working as a whole. You will understand how the different elements work and how to configure and tweak them.

Technical requirements

For this chapter, you need to have a local Kubernetes instance running as described in the previous chapter. Remember to have the Ingress controller installed.

You can check the full code that we are going to use in the GitHub repository (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter06`).

Implementing multiple services

In the GitHub repo, you can find the three microservices that we will be using in this chapter. They are based on the monolith introduced in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, and are split into three elements:

- **Thoughts Backend**: As described in the previous chapter, this handles the storage of thoughts and the search for them.
- **Users Backend**: This stores the users and allows them to log in. Based on the description of the authentication method, this creates a token that can be used to authenticate against other systems.
- **Frontend**: This comes from the monolith, but instead of accessing a database directly, it makes requests to the User and Thoughts Backends to replicate the functionality.

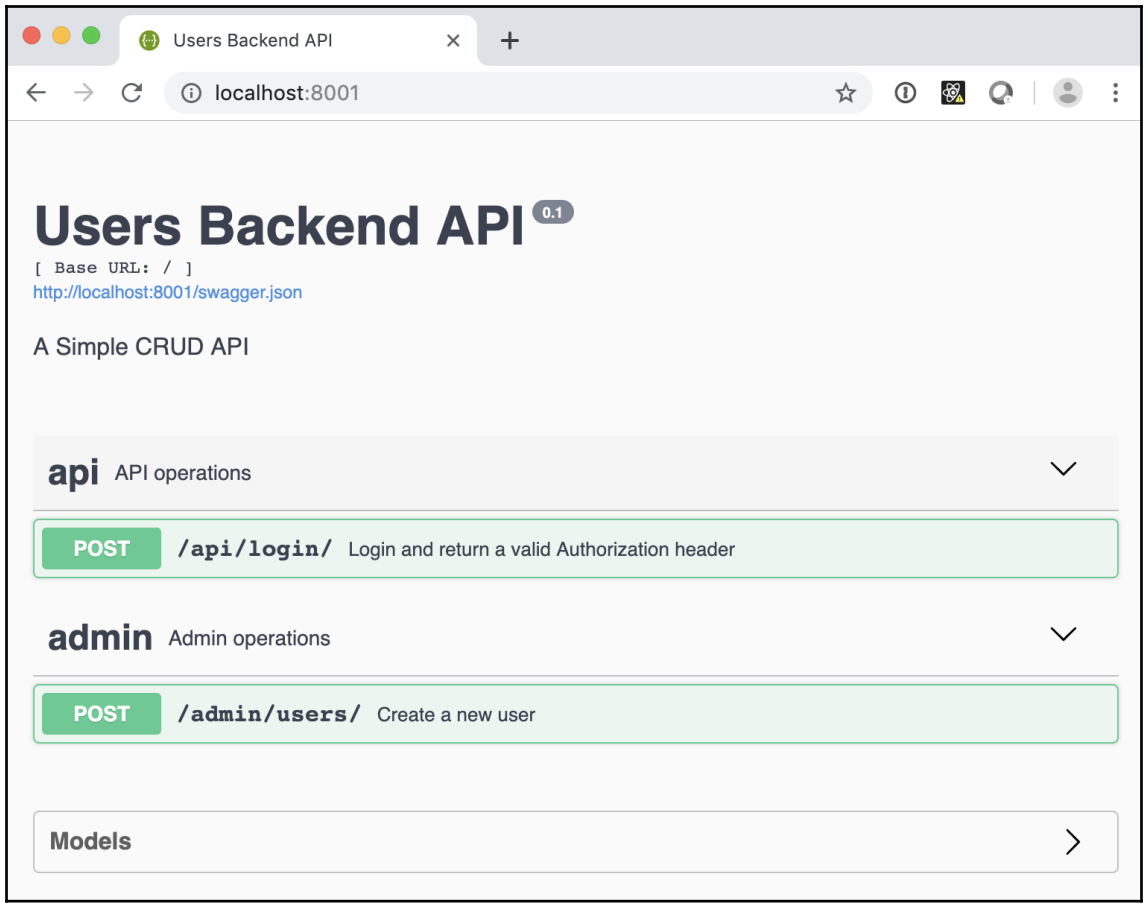
Note that the static files are still being served by the Frontend, even though we described the final stage of the cluster serving them independently. This is done for simplicity and to avoid having an extra service.

The aforementioned services are Dockerized in similar ways to how the Thoughts Backend was in `Chapter 3`, *Build, Run, and Test Your Service Using Docker*. Let's look at some of the details for the other microservices.

Describing the Users Backend microservice

The code for the Users Backend can be found at `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter06/users_backend`. The structure is very similar to the Thoughts Backend, a Flask-RESTPlus application that communicates to a PostgreSQL database.

It has two endpoints, as seen in its Swagger interface:



The endpoints are as follows:

	Endpoint	Input	Returns
POST	/api/login	{username: <username>, password: <password>}	{Authorized: <token header>}
POST	/admin/users	{username: <username>, password: <password>}	<new_user>

The `admin` endpoint allows you to create new users, and the login API returns a valid header that can be used for the Thoughts Backend.

The users are stored in the database with the following schema:

Field	Format	Comments
id	Integer	Primary key
username	String (50)	Username
password	String(50)	Password stored in plain text, which is a bad idea, but simplifies the example
creation	Datetime	The time of the creation of the user

This schema, in SQLAlchemy model definition, is described using the following code:

```
class UserModel(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50))
    # DO NOT EVER STORE PLAIN PASSWORDS IN DATABASES
    # THIS IS AN EXAMPLE!!!!!
    password = db.Column(db.String(50))
    creation = db.Column(db.DateTime, server_default=func.now())
```

Note that the creation date gets stored automatically. Also, note that we store the password in plain text. This is a *terrible, terrible idea in a production service*. You can check out an article called *How to store a password in the database?* (`https://www.geeksforgeeks.org/store-password-database/`) to get general ideas for encrypting passwords with a salt seed. You can use a package such as `pyscrypt` (`https://github.com/ricmoo/pyscrypt`) to implement this kind of structure in Python.

The users *bruce* and *stephen* are added to the `db` example as a way of having example data.

Describing the Frontend microservice

The Frontend code is available in the GitHub repo. It is based on the Django monolith (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter01/Monolith`) introduced in `Chapter 1`, *Making the Move – Design, Plan, and Execute*.

The main difference from the monolith is that the database is not accessed. Therefore, there are no uses for the Django ORM. They are replaced with HTTP requests to the other backends. To make the requests, we use the fantastic `requests` library.

For example, the `search.py` file gets converted into the following code, which delegates the search toward the Thoughts Backend microservice. Note how the request by the customer gets transformed into an internal API call to the `GET /api/thoughts` endpoint. The result is decoded in JSON and rendered in the template:

```
import requests

def search(request):
    username = get_username_from_session(request)
    search_param = request.GET.get('search')

    url = settings.THOUGHTS_BACKEND + '/api/thoughts/'
    params = {
        'search': search_param,
    }
    result = requests.get(url, params=params)
    results = result.json()

    context = {
        'thoughts': results,
        'username': username,
    }
    return render(request, 'search.html', context)
```

The monolith equivalent code can be compared in the `Chapter01` subdirectory of the repo (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter01/Monolith/mythoughts/thoughts/search.py`).

Note how we make a `get` request through the `requests` library to the defined search endpoint, which results in the `json` format being returned and rendered.

The `THOUGTHS_BACKEND` root URL comes from the settings, in usual Django fashion.

This example is a simple one because there's no authentication involved. The parameters are captured from the user interface, then routed toward the backend. The request gets properly formatted both toward the backend and once the result is obtained, and then rendered. This is the core of two microservices working together.

A more interesting case is the `list_thought` (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter06/frontend/mythoughts/thoughts/thoughts.py#L18`) view. The following code lists the thoughts for the logged in user:

```
def list_thoughts(request):
    username = get_username_from_session(request)
    if not username:
        return redirect('login')

    url = settings.THOUGHTS_BACKEND + '/api/me/thoughts/'
    headers = {
        'Authorization': request.COOKIES.get('session'),
    }
    result = requests.get(url, headers=headers)
    if result.status_code != http.client.OK:
        return redirect('login')

    context = {
        'thoughts': result.json(),
        'username': username,
    }
    return render(request, 'list_thoughts.html', context)
```

Here, before doing anything, we need to check whether a user is logged in. This is done in the `get_username_from_session` call, which returns the `username` or `None`, if they're not logged in. If they're not logged in, the return gets redirected to the login screen.

As this endpoint requires authentication, we need to add the session from the user in an `Authorization` header to our request. The session of the user can be obtained from the `request.COOKIES` dictionary.

As a safeguard, we need to check whether the returning status code from the backend is correct. For this call, any resulting status code that's not a 200 (HTTP call correct) will produce a redirection to the login page.

For simplicity and clarity, our example services are not handling different error cases. In a production system, there should be a differentiation between errors where the issue is that either the user is not logged in or there's another kind of user error (a 400 error), or the backend service is not available (a 500 status code).

Error handling, when done properly, is difficult, but worth doing well, especially if the error helps users to understand what happened.

The `get_username_from_session` function encapsulates a call to `validate_token_header`, the same one as introduced in the previous chapter:

```
def get_username_from_session(request):
    cookie_session = request.COOKIES.get('session')
    username = validate_token_header(cookie_session,
                                     settings.TOKENS_PUBLIC_KEY)
    if not username:
        return None

    return username
```

The `settings` file contains the public key required to decode the token.

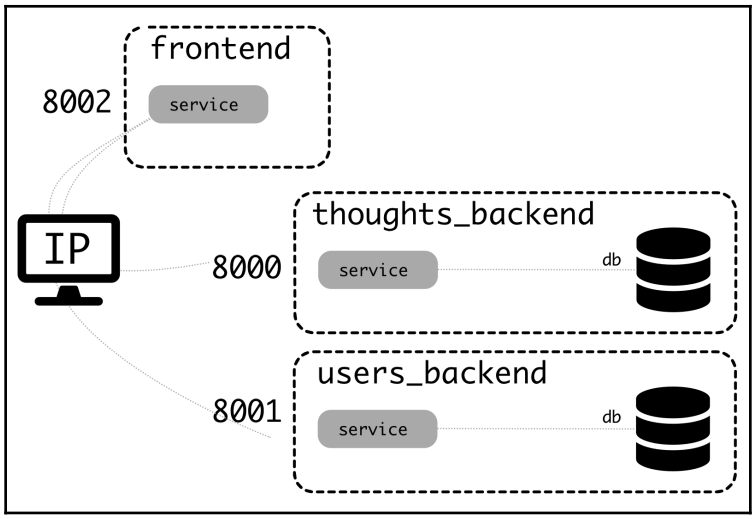
In this chapter, for simplicity, we copied the key directly into the `settings` file. This is not the way to go for a production environment. Any secret should be obtained through the Kubernetes environment configuration. We will see how to do this in the following chapters.

The environment file needs to specify where both the base URLs for the Users Backend and the Thoughts Backend are, to be able to connect to them.

Connecting the services

It's possible to test the services working in unison only with `docker-compose`. Check that the `docker-compose.yaml` files in both the Users Backend and the Thoughts Backend expose different ports externally.

The Thoughts Backend exposes port `8000` and the Users Backend exposes port `8001`. This allows the Frontend to connect to them (and expose port `8002`). This diagram shows how this system works:



You can see how the three services are isolated, as `docker-compose` will create its own network for them to connect. Both backends have their own container, which acts as the database.

The Frontend service needs to connect to the others. The URL of the services should be added to the `environment.env` file and should indicate the service with the IP of the computer.

An internal IP such as localhost or `127.0.0.1` does not work, as it gets interpreted **inside the container.** You can obtain the local IP by running `ifconfig`.

For example, if your local IP is `10.0.10.3`, the `environment.env` file should contain the following:

```
THOUGHTS_BACKEND_URL=http://10.0.10.3:8000
USER_BACKEND_URL=http://10.0.10.3:8001
```

If you access the Frontend service in your browser, it should connect to the other services.

A possibility could be to generate a bigger `docker-compose` file that includes everything. This could make sense if all the microservices are in the same Git repo, a technique known as **monorepo** (`https://gomonorepo.org/`). Possible problems include keeping both the internal `docker-compose` to work with a single system and the general one in sync so that the automated tests should detect any problems.

This structure is a bit cumbersome, so we can transform it into a proper Kubernetes cluster, aiming at local development.

Configuring the services

To configure the apps in Kubernetes, we need to define the following Kubernetes objects per app:

- **Deployment**: The deployment will control the creation of pods, so they will always be available. It will also create them based on the image and will add configuration, where needed. The pod runs the app.
- **Service**: The service will make the RESTful requests available inside the cluster, with a short name. This routes the requests to any available pod.
- **Ingress**: This makes the service available outside of the cluster, so we can access the app from outside the cluster.

In this section, we will look at the Thoughts Backend configuration in detail as an example. Later, we will see how the different parts connect. We created a Kubernetes sub-directory (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter06/thoughts_backend/kubernetes`) to store the `.yaml` files with each of the definitions.

We will use the `example` namespace, so be sure that it's created:

```
$ kubectl create namespace example
```

Let's start with the first Kubernetes object.

Configuring the deployment

For the Thoughts Backend deployment, we will deploy a pod with two containers, one with the database, and another with the application. This configuration makes it easy to work locally but keep in mind that recreating the pod will restart both containers.

The file for configuration is fully available here (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter06/thoughts_backend/kubernetes/deployment.yaml`), so let's take a look at its different parts. The first element describes what it is and its name, as well as the namespace it lives at:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: thoughts-backend
    labels:
        app: thoughts-backend
    namespace: example
```

Then, we generate `spec`. It contains how many pods we should keep and the template for each pod. `selector` defines what labels are monitored, and it should match the `labels` in the template:

```
spec:
    replicas: 1
    selector:
        matchLabels:
            app: thoughts-backend
```

The `template` section defines the containers in its own `spec` section:

```
    template:
        metadata:
            labels:
                app: thoughts-backend
        spec:
            containers:
                - name: thoughts-backend-service
                  ...
                - name: thoughts-backend-db
                  ...
```

`thoughts-backend-db` is simpler. The only required element is to define the name of the container and the image. We need to define the pulling policy as `Never` to indicate that the image is available in the local Docker repo, and that it's not necessary to pull it from a remote registry:

```
- name: thoughts-backend-db
  image: thoughts_backend_db:latest
  imagePullPolicy: Never
```

`thoughts-backend-service` needs to define the exposed port for the service as well as the environment variables. The variable values are the ones that we used previously when creating the database, except for `POSTGRES_HOST`, where we have the advantage that all containers in the same pod share the same IP:

```
- name: thoughts-backend-service
  image: thoughts_server:latest
  imagePullPolicy: Never
  ports:
  - containerPort: 8000
  env:
  - name: DATABASE_ENGINE
    value: POSTGRESQL
  - name: POSTGRES_DB
    value: thoughts
  - name: POSTGRES_USER
    value: postgres
  - name: POSTGRES_PASSWORD
    value: somepassword
  - name: POSTGRES_PORT
    value: "5432"
  - name: POSTGRES_HOST
    value: "127.0.0.1"
```

To get the deployment in Kubernetes, you need to apply the file, as shown here:

```
$ kubectl apply -f thoughts_backend/kubernetes/deployment.yaml
deployment "thoughts-backend" created
```

The deployment is now created in the cluster:

```
$ kubectl get deployments -n example
NAME              DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
thoughts-backend 1       1       1          1         20s
```

This creates pods automatically. If the pod is deleted or crashes, the deployment will restart it with a different name:

```
$ kubectl get pods -n example
NAME                                READY STATUS  RESTARTS AGE
thoughts-backend-6dd57f5486-l9tgg 2/2   Running 0        1m
```

The deployment is tracking the latest image, but it won't create a new pod unless it's deleted. To make changes, be sure to delete the pod manually, after which it will be recreated:

```
$ kubectl delete pod thoughts-backend-6dd57f5486-l9tgg -n example
pod "thoughts-backend-6dd57f5486-l9tgg" deleted
$ kubectl get pods -n example
NAME                                READY STATUS  RESTARTS AGE
thoughts-backend-6dd57f5486-nf2ds 2/2   Running 0        28s
```

The application is still not discoverable inside the cluster, other than referring to it by its specific pod name, which can change, so we need to create a service for that.

Configuring the service

We create a Kubernetes service to create a name for the application exposed by the created deployment. The service can be checked in the `service.yaml` file. Let's take a look:

```
---
apiVersion: v1
kind: Service
metadata:
    namespace: example
    labels:
        app: thoughts-service
    name: thoughts-service
spec:
    ports:
        - name: thoughts-backend
          port: 80
          targetPort: 8000
    selector:
        app: thoughts-backend
    type: NodePort
```

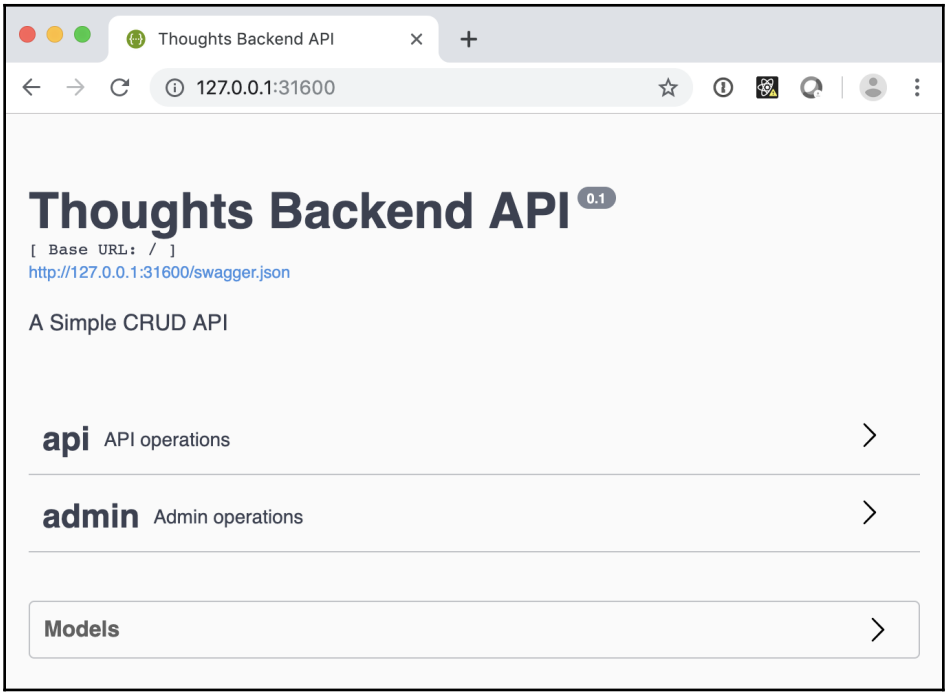
The initial data is similar to the deployment. The `spec` section defines the open ports, routing access to the service on port `80` to port `8000` in containers in `thoughts-backend`, the name of the deployment. The `selector` part routes all the requests to any pod that matches.

The type is `NodePort` to allow access from outside the cluster. This allows us to check that it is working correctly, once we find the externally exposed IP:

```
$ kubectl apply -f kubernetes/service.yaml
service "thoughts-service" configured
```

```
$ kubectl get service -n example
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
thoughts-service 10.100.252.250 <nodes> 80:31600/TCP 1m
```

We can access the Thoughts Backend by accessing localhost with the described pod. In this case, `http://127.0.0.1:31600`:



The service gives us an internal name, but if we want to have control over how it is exposed externally, we need to configure an Ingress.

Configuring the Ingress

Finally, we describe the Ingress in `ingress.yaml` (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter06/thoughts_backend/kubernetes/ingress.yaml`). The file is copied here. Note how we set up the metadata to live in the proper namespace:

```
---
apiVersion: extensions/v1beta1
kind: Ingress
```

```
metadata:
    name: thoughts-backend-ingress
    namespace: example
spec:
    rules:
        - host: thoughts.example.local
          http:
            paths:
              - backend:
                  serviceName: thoughts-service
                  servicePort: 80
                path: /
```

This Ingress will make the service be exposed to the nodes on port `80`. As multiple services can be exposed on the same nodes, they get distinguished by their hostname, in this case, `thoughts.example.local`.

The Ingress controller we are using only allows exposing ports `80` (HTTP) and `443` (HTTPS) in `servicePort`.

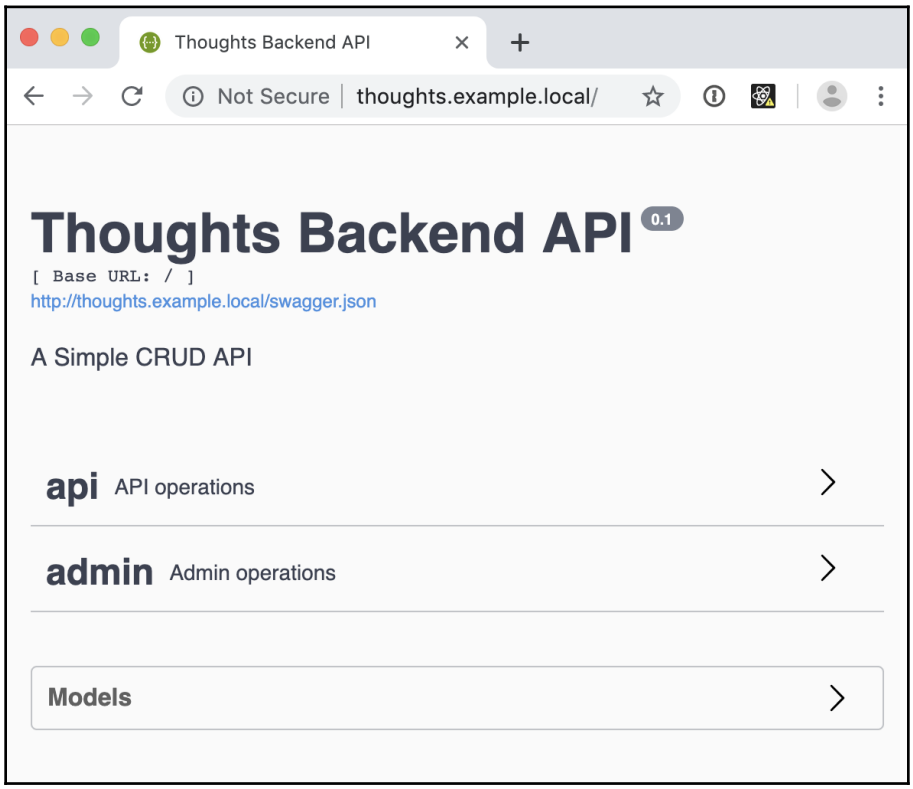
After applying the service, we can try to access the page, but, unless we address the calls toward the proper host, we will get a 404 error:

```
$ kubectl apply -f kubernetes/ingress.yaml
ingress "thoughts-backend-ingress" created
$ kubectl get ingress -n example
NAME                      HOSTS                     ADDRESS   PORTS   AGE
thoughts-backend-ingress thoughts.example.local localhost 80 1m
$ curl http://localhost
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx/1.15.8</center>
</body>
</html>
```

We need to be able to point any request to `thoughts.example.local` to our localhost. In Linux and macOS, the easiest way is to change your `/etc/hosts` file to include the following line:

```
127.0.0.1 thoughts.example.local
```

Then, we can use a browser to check our application, this time in `http://thoughts.example.local` (and port `80`):



Defining different host entries allows us to access all the services externally, to be able to tweak them and debug problems. We will define the rest of the Ingresses in the same way.

If you get a `Connection refused` error and the word `localhost` does not appear when running `kubectl get ingress -n example`, your Kubernetes installation does not have the Ingress controller installed. Double-check the installation documentation at `https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md`.

So now we have a working application deployed in Kubernetes locally!

Deploying the full system locally

Each of our microservices works on its own, but to have the whole system working, we need to deploy the three of them (Thoughts Backend, Users Backend, and Frontend) and connect them to each other. The Frontend, in particular, requires the other two microservices to be up and running. With Kubernetes, we can deploy it locally.

To deploy the full system, we need to deploy the Users Backend first, and then the Frontend. We will describe each of these systems, relating them to the already deployed Thoughts Backend, which we saw how to deploy before.

Deploying the Users Backend

The Users Backend files are very similar to the Thoughts Backend. You can check them in the GitHub repo (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter06/users_backend/kubernetes`). Be sure that the environment settings in the `deployment.yaml` values are correct:

```
$ kubectl apply -f users_backend/kubernetes/deployment.yaml
deployment "users-backend" created
$ kubectl apply -f users_backend/kubernetes/service.yaml
service "users-service" created
$ kubectl apply -f users_backend/kubernetes/ingress.yaml
ingress "users-backend-ingress" created
```

Remember to be sure to include the new hostname in `/etc/hosts`:

```
127.0.0.1 users.example.local
```

You can access the Users Backend in `http://users.example.local`.

Adding the Frontend

The Frontend service and Ingress are very similar to the previous ones. The deployment is slightly different. Let's take a look at the configuration, in three groups:

1. First, we add the metadata about the `namespace`, `name`, and the `kind` (deployment) as shown in the following code:

```
---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
    name: frontend
    labels:
        app: frontend
    namespace: example
```

2. Then, we define the `spec` with the template and the number of `replicas`. Only one replica is fine for a local system:

```
spec:
    replicas: 1
    selector:
        matchLabels:
            app: frontend
    template:
        metadata:
            labels:
                app: frontend
```

3. Finally, we `spec` out the template with the container definition:

```
spec:
    containers:
        - name: frontend-service
          image: thoughts_frontend:latest
          imagePullPolicy: Never
          ports:
             - containerPort: 8000
          env:
              - name: THOUGHTS_BACKEND_URL
                value: http://thoughts-service
              - name: USER_BACKEND_URL
                value: http://users-service
```

The main difference from the previously defined Thoughts Backend deployment is that there's a single container and that the environment on it is simpler.

We define the backend URLs environments as the service endpoints. These endpoints are available inside the cluster, so they'll be directed to the proper containers.

Remember that the `*.example.local` addresses are only available in your computer, as they only live in `/etc/hosts`. Inside the container, they won't be available.

This is suitable for local development, but an alternative is to have a DNS domain that can be redirected to `127.0.0.1` or similar.

We should add a new domain name in the `/etc/hosts` file:

```
127.0.0.1 frontend.example.local
```

Django requires you to set up the `ALLOWED_HOSTS` setting's value, to allow it to accept the hostname, as, by default, it only allows connections from localhost. See the Django documentation (`https://docs.djangoproject.com/en/2.2/ref/settings/#allowed-hosts`) for more information. To simplify things, we can allow any host using `'*'`. Check out the code on GitHub (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter06/frontend/mythoughts/mythoughts/settings.py#L28`).

In production, it's good practice to limit the hosts to the **Fully Qualified Domain Name** (**FQDN**), the full DNS name of a host, but the Kubernetes Ingress will check the host header and reject it if it's not correct.

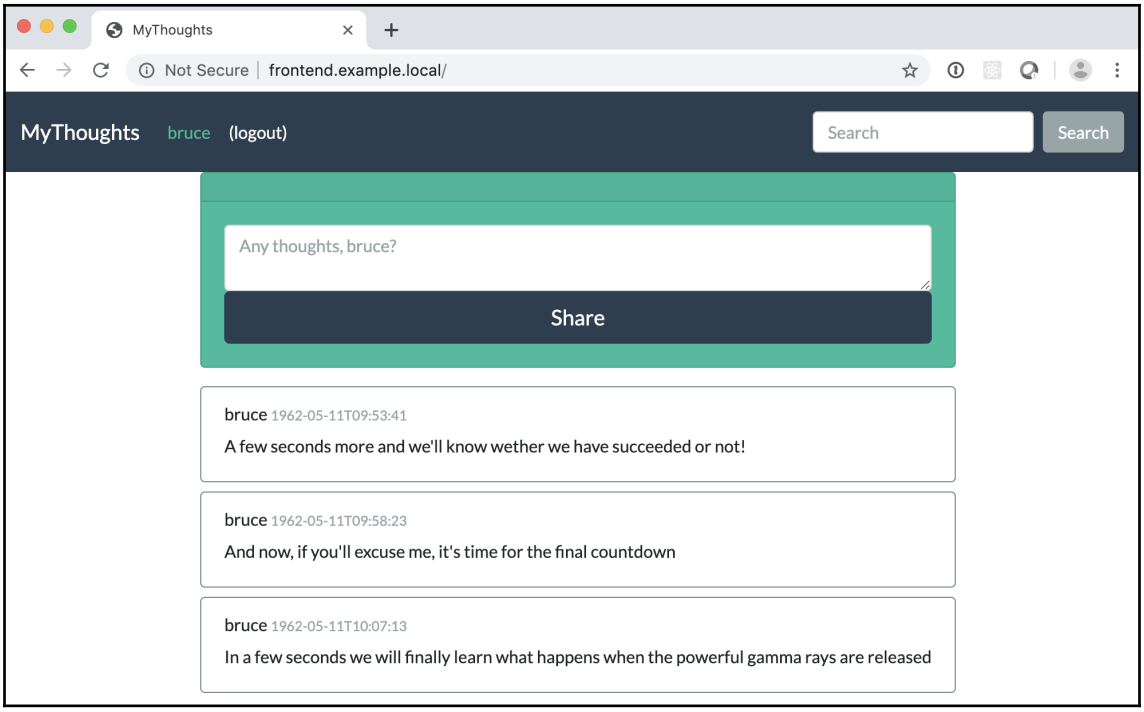
The Frontend application gets deployed as we've done before:

```
$ kubectl apply -f frontend/kubernetes/deployment.yaml
deployment "frontend" created
$ kubectl apply -f frontend/kubernetes/service.yaml
service "frontend-service" created
$ kubectl apply -f frontend/kubernetes/ingress.yaml
ingress "frontend-ingress" created
```

Then we can access the full system, login, search, and so on.

Remember that there are two users, `bruce` and `stephen`. Their passwords are the same as their usernames. You don't need to be logged in to search.

In your browser, go to `http://frontend.example.local/`:



Congratulations! You have a working Kubernetes system, including different deployed microservices. You can access each of the microservices independently to debug it or to carry out actions such as creating a new user, and so on.

If you need to deploy a new version, build the proper containers using the `docker-compose` build and delete the pod to force the recreation of it.

Summary

In this chapter, we saw how to deploy our microservices in a Kubernetes local cluster to allow local development and testing. Having the whole system deployed on your local computer greatly simplifies developing new features or debugging the behavior of the system. The production environment will be very similar, so this also lays the foundation for it.

We first described the two microservices that were missing. The Users Backend handles the authentication for users and Frontend is a modified version of the monolith presented in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, which connects to the two backends. We showed how to build and run them in a `docker-compose` way.

After that, we described how to set up a combination of `.yaml` files to configure applications properly in Kubernetes. Each microservice has its own deployment to define the available pods, a service to define a stable access point, and an Ingress to allow external access. We described them in detail, and then applied them to all of the microservices.

In the next chapter, we will see how to move from local deployment and deploy a Kubernetes cluster ready for production.

Questions

1. What are the three microservices that we are deploying?
2. Which microservice requires the other two to be available?
3. Why do we need to use external IPs to connect the microservices while running in `docker-compose`?
4. What are the main Kubernetes objects required for each application?
5. Are any of the objects not required?
6. Can you see any issues if we scale any of the microservices to more than one pod?
7. Why are we using the `/etc/hosts` file?

Further reading

You can learn more about Kubernetes in the books *Kubernetes for Developers* (`https://www.packtpub.com/eu/virtualization-and-cloud/kubernetes-developers`) and *Kubernetes Cookbook - Second Edition* (`https://www.packtpub.com/in/virtualization-and-cloud/kubernetes-cookbook-second-edition`).

7
Configuring and Securing the Production System

Production (from production environment) is a common name to describe the main system—the one that does the work for the real customers. This is the main environment available in the company. It can also be called **live**. This system needs to be openly available on the internet to be useful, which also makes security and reliability a big priority. In this chapter, we'll see how to deploy a Kubernetes cluster for production.

We'll see how to set one up using a third-party offering **Amazon Web Services** (**AWS**), and will cover why it's a bad idea to create your own. We will deploy our system in this new deployment, and will check how to set up a load balancer to move traffic from the old monolith to the new system in an ordered fashion.

We will also see how to scale automatically both the pods inside the Kubernetes cluster and the nodes to adapt the resources to the needs.

The following topics will be covered in this chapter:

- Using Kubernetes in the wild
- Setting up the Docker registry
- Creating the cluster
- Using HTTPS and TLS to secure external access
- Being ready for migration to microservices
- Autoscaling the cluster
- Deploying a new Docker image smoothly

We will also cover some good practices to ensure that our deployments get deployed as smoothly and reliably as possible. By the end of the chapter, you'll have the system deployed in a publicly available Kubernetes cluster.

Technical requirements

We will use AWS as our cloud vendor for the examples. We need to install some utilities to interact from the command line. Check how to install the AWS CLI utility in this documentation (`https://aws.amazon.com/cli/`). This utility allows performing AWS tasks from the command line.

To operate the Kubernetes cluster, we will use `eksctl`. Check this documentation (`https://eksctl.io/introduction/installation/`) for installation instructions.

You'll need also to install `aws-iam-authenticator`. You can check the installation instructions here (`https://docs.aws.amazon.com/eks/latest/userguide/install-aws-iam-authenticator.html`).

The code for this chapter can be found on GitHub at this link: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter07`.

Be sure you have `ab` (Apache Bench) installed on your computer. It is bundled with Apache, and it's installed by default in macOS and some Linux distributions. You can check this article: `https://www.petefreitag.com/item/689.cfm`.

Using Kubernetes in the wild

When deploying a cluster to be used as production, the best possible advice is to use a commercial service. All the main cloud providers (AWS EKS, **Google Kubernetes Engine** (**GKE**), and **Azure Kubernetes Service** (**AKS**)) allow you to create a managed Kubernetes cluster, meaning that the only required parameter is to choose the number and type of physical nodes and then access it through `kubectl`.

We will use AWS for the examples in this book, but take a look at the documentation of other providers in case they work better for your use case.

Kubernetes is an abstraction layer, so this way of operation is very convenient. The pricing is similar to paying for raw instances to act as node servers and removes the need to install and manage the Kubernetes Control Plane so the instances act as Kubernetes nodes.

It's worth saying it again: unless you have a very good reason, *do not deploy your own Kubernetes cluster*; instead, use a cloud provider offering. It will be easier and will save you from a lot of maintenance costs. Configuring a Kubernetes node in a way that's performant and implements good practices to avoid security problems is not trivial.

Creating your own Kubernetes cluster may be unavoidable if you have your own internal data center, but in any other case it makes much more sense to directly use one managed by a known cloud provider. Probably your current provider already has an offering for managed Kubernetes!

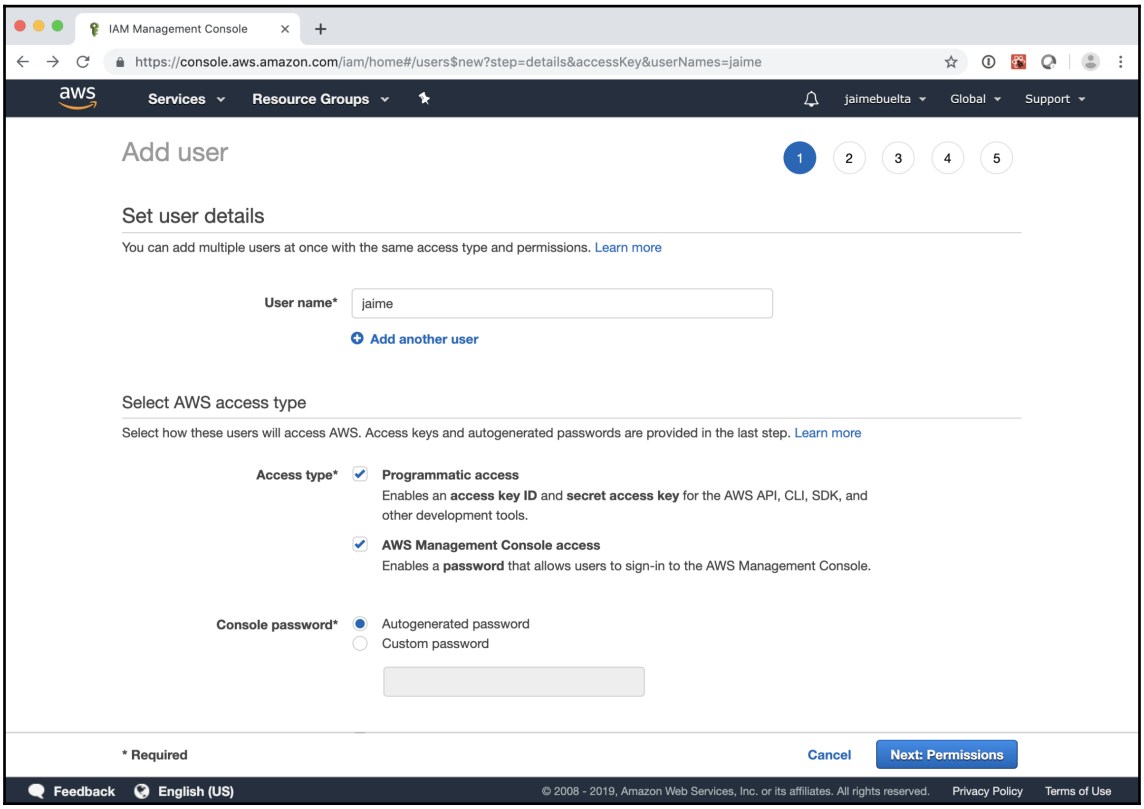
Creating an IAM user

AWS works using different users to grant them several roles. They carry different permissions that enable the users to perform actions. This system is called **Identity and Access Management** (**IAM**) in AWS nomenclature.

Creating a proper IAM user could be quite complicated, depending on your settings and how AWS is used in your organization. Check the documentation (`https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html`) and find the people responsible for dealing with AWS in your organization and check with them to see what the required steps are.

Let's take a look at the steps to create an IAM user:

1. We need to create an AWS user if it is not created with proper permissions. Be sure that it will be able to access the API by activating the **Programmatic access** as seen in the following screenshot:



   This will show its **Access Key**, **Secret Key**, and **Password**. Be sure to store them securely.

2. To access through the command line, you need to use the AWS CLI. With the AWS CLI and the access information, configure your command line to use `aws`:

```
$ aws configure
AWS Access Key ID [None]: <your Access Key>
AWS Secret Access Key [None]: <your Secret Key>
Default region name [us-west-2]: <EKS region>
Default output format [None]:
```

You should be able to get the identity to check that the configuration is successful using the following command:

```
$ aws sts get-caller-identity
{
    "UserId": "<Access Key>",
    "Account": "<account ID>",
    "Arn": "arn:aws:iam::XXXXXXXXXXX:user/jaime"
}
```

You can now access the command-line AWS actions.

Keep in mind that the IAM user can create more keys if necessary, revoke the existing ones, and so on. This normally is handled by an admin user in charge of AWS security. You can read more in the Amazon documentation (`https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html#Using_CreateAccessKey_API`). Key rotation is a good idea to ensure that old keys are deprecated. You can do it through the `aws` client interface.

We will use the web console for some operations, but others require the usage of `aws`.

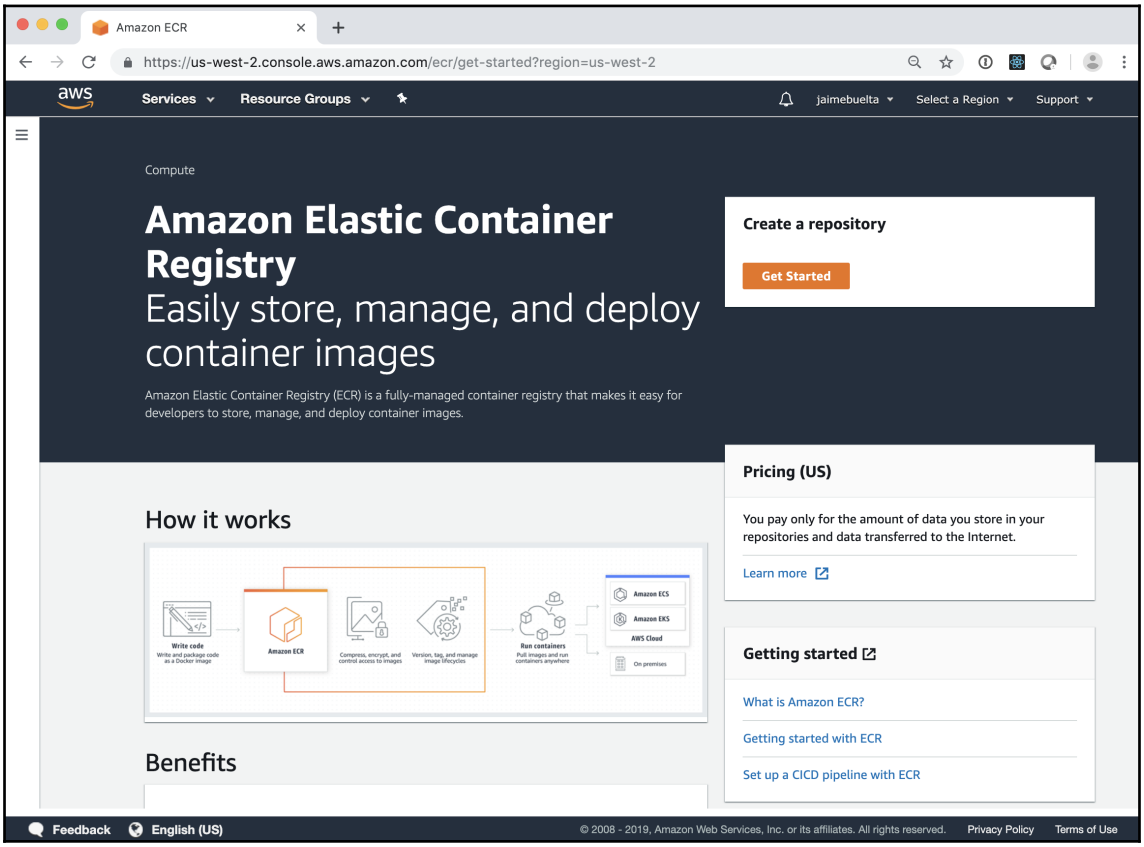
Setting up the Docker registry

We need to be able to reach the Docker registry where the images to be deployed are stored. The easiest way of ensuring that the Docker registry is reachable is to use the Docker registry in the same service.

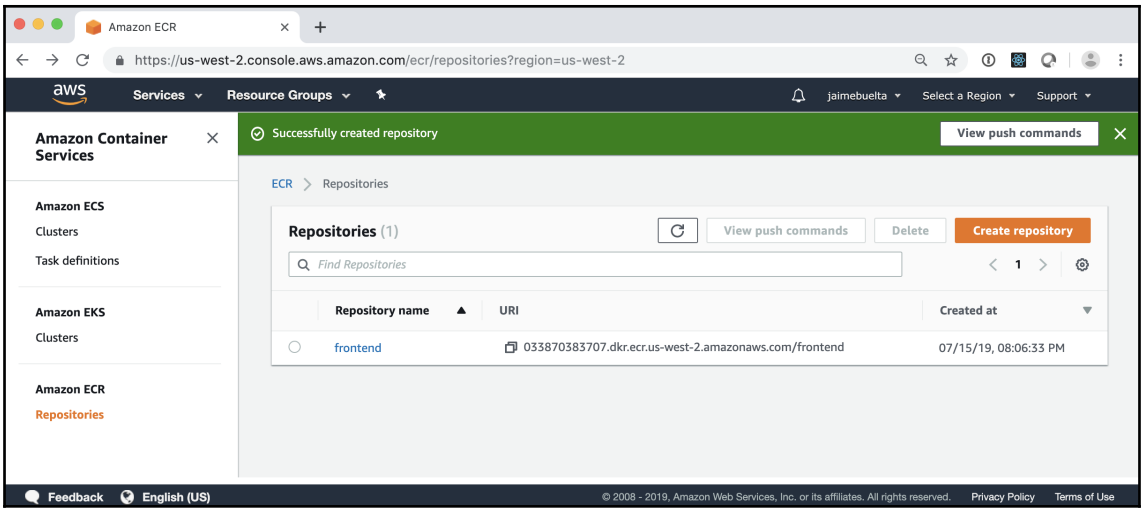
You can still use the Docker Hub registry, but using a registry in the same cloud provider is typically easier as it's better integrated. It will also help in terms of authentication.

We need to configure an **Elastic Container Registry** (**ECR**), using the following steps:

1. Log into the AWS console and search for Kubernetes or ECR:



2. Create a new registry called `frontend`. It will create a full URL, which you will need to copy:



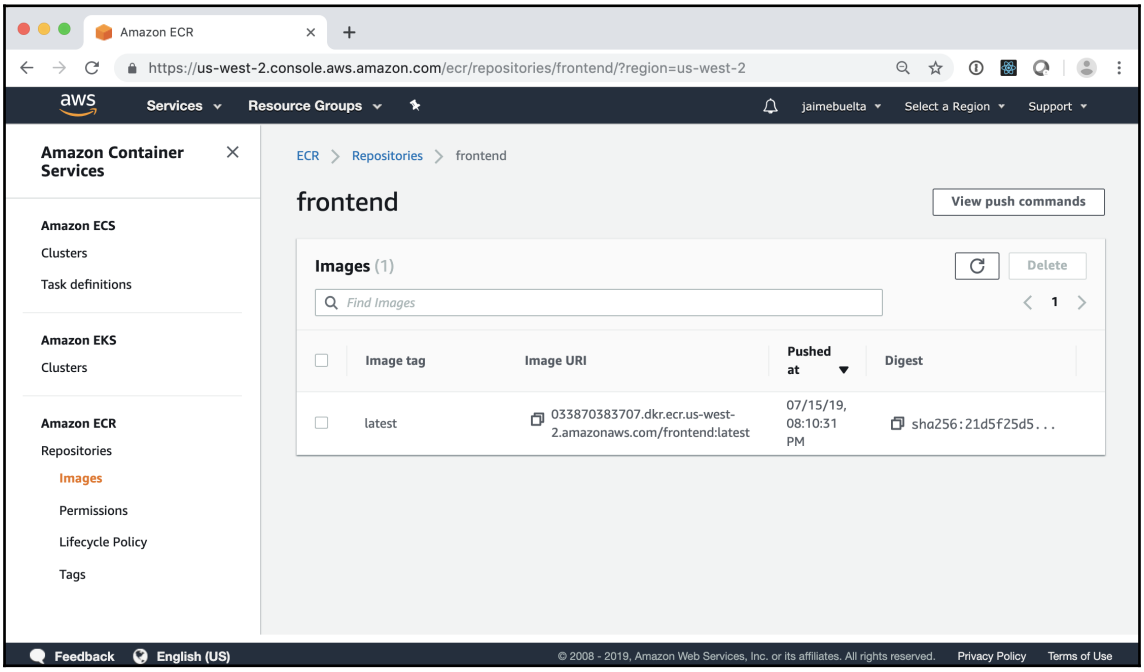
3. We need to make our local `docker` log in the registry. Note that `aws ecr get-login` will return a `docker` command that will log you in, so copy it and paste:

```
$ aws ecr get-login --no-include-email
<command>
$ docker login -u AWS -p <token>
Login Succeeded
```

4. Now we can tag the image that we want to push with the full registry name, and push it:

```
$ docker tag thoughts_frontend 033870383707.dkr.ecr.us-
west-2.amazonaws.com/frontend
$ docker push 033870383707.dkr.ecr.us-west-2.amazonaws.com/frontend
The push refers to repository [033870383707.dkr.ecr.us-
west-2.amazonaws.com/frontend]
...
latest: digest:
sha256:21d5f25d59c235fe09633ba764a0a40c87bb2d8d47c7c095d254e20f7b43
7026 size: 2404
```

5. The image is pushed! You can check it by opening the AWS console in the browser:

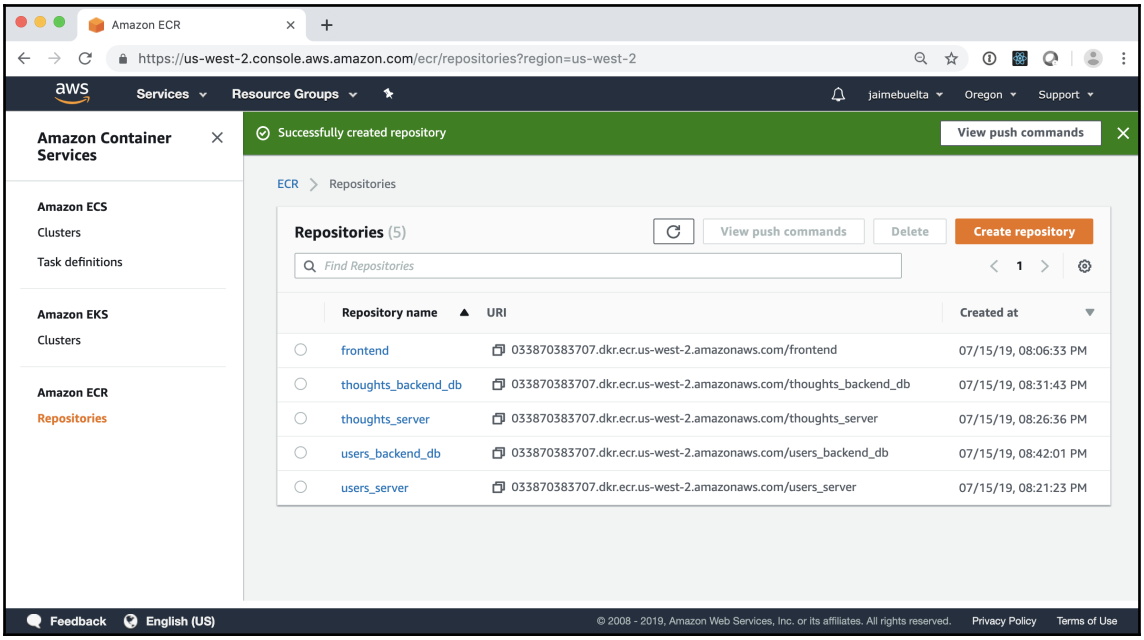


6. We need to repeat the process to also push the Users Backend and Thoughts Backend.

We use the setting of two containers for the deployment of the Users Backend and Thoughts Backend, which includes one for the service and another for a volatile database. This is done for demonstration purposes, but won't be the configuration for a production system, as the data will need to be persistent.

At the end of the chapter, there's a question about how to deal with this situation. Be sure to check it!

All the different registries will be added. You can check them in the browser AWS console:



Our pipelines will need to be adapted to push to this repositories.

A good practice in deployment is to make a specific step called **promotion**, where the images ready to use in production are copied to an specific registry, lowering the chance that a bad image gets deployed by mistake in production.

This process may be done several times to promote the images in different environments. For example, deploy a version in an staging environment. Run some tests, and if they are correct, promote the version, copying it into the production registry and labelling it as good to deploy on the production environment.

This process can be done with different registries in different providers.

We need to use the name of the full URL in our deployments.

Creating the cluster

To make our code available in the cloud and publicly accessible, we need to set up a working production cluster, which requires two steps:

1. Create the EKS cluster in AWS cloud (this enables you to run the `kubectl` commands that operate in this cloud cluster).
2. Deploy your services, using a set of `.yaml` files, as we've seen in previous chapters. The files require minimal changes to adapt them to the cloud.

Let's check the first step.

Creating the Kubernetes cluster

The best way to create a cluster is to use the `eksctl` utility. This automates most of the work for us, and allows us to scale it later if we have to.

Be aware that EKS is available only in some regions, not all. Check the AWS regional table (`https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/`) to see the available zones. We will use the Oregon (`us-west-2`) region.

To create the Kubernetes cluster, let's take the following steps:

1. First, check that `eksctl` is properly installed:

```
$ eksctl get clusters
No clusters found
```

2. Create a new cluster. It will take around 10 minutes:

```
$ eksctl create cluster -n Example
[i] using region us-west-2
[i] setting availability zones to [us-west-2d us-west-2b us-
west-2c]
...
[✔]  EKS cluster "Example" in "us-west-2" region is ready
```

3. This creates the cluster. Checking the AWS web interface will show the newly configured elements.

The `--arg-access` option needs to be added for a cluster capable of autoscaling. This will be described in more detail in the *Autoscaling the cluster* section.

4. The `eksctl create` command also adds a new context with the information about the remote Kubernetes cluster and activates it, so `kubectl` will now point to this new cluster.

Note that `kubectl` has the concept of contexts as different clusters it can connect. You can see all the available contexts running `kubectl config get-contexts` and `kubectl config use-context <context-name>` to change them. Check the Kubernetes documentation (`https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/`) on how to create new contexts manually.

5. This command sets `kubectl` with the proper context to run commands. By default, it generates a cluster with two nodes:

```
$ kubectl get nodes
NAME                       STATUS ROLES AGE VERSION
ip-X.us-west-2.internal Ready <none> 11m v1.13.7-eks-c57ff8
ip-Y.us-west-2.internal Ready <none> 11m v1.13.7-eks-c57ff8
```

6. We can scale the number of nodes. To reduce the usage of resources and save money. We need to retrieve the name of the nodegroup, which controls the number of nodes, and then downscale it:

```
$ eksctl get nodegroups --cluster Example
CLUSTER NODEGROUP CREATED MIN SIZE MAX SIZE DESIRED CAPACITY
INSTANCE TYPE IMAGE ID
Example ng-fa5e0fc5 2019-07-16T13:39:07Z 2 2 0 m5.large
ami-03a55127c613349a7
$ eksctl scale nodegroup --cluster Example --name ng-fa5e0fc5 -N 1
[i] scaling nodegroup stack "eksctl-Example-nodegroup-ng-fa5e0fc5"
in cluster eksctl-Example-cluster
[i] scaling nodegroup, desired capacity from to 1, min size from 2
to 1
```

7. You can contact the cluster through `kubectl` and carry the operations normally:

```
$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.100.0.1 <none> 443/TCP 7m31s
```

The cluster is set up, and we can run commands on it from our command line.

Creating an EKS cluster can be tweaked in a lot of ways, but AWS can be temperamental in terms of access, users, and permissions. For example, the cluster likes to have a CloudFormation rule to handle the cluster, and all the elements should be created with the same IAM user. Check with anyone that works with the infrastructure definition in your organization to check what's the proper configuration. Don't be afraid of running tests, a cluster can be quickly removed through the `eksctl` configuration or the AWS console.

Furthermore, `eksctl` creates the cluster with the nodes in different availability zones (AWS isolated locations within the same geographical region) wherever possible, which minimizes the risk of getting the whole cluster down because of problems in AWS data centers.

Configuring the cloud Kubernetes cluster

The next stage is to run our services on the EKS cluster, so it's available in the cloud. We will use the `.yaml` files as a base, but we need to make a few changes.

Take a look at the files in the GitHub `Chapter07` (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter07`) subdirectory.

We will see the differences from the Kubernetes configuration files in the previous chapter, and then we will deploy them in the *Deploying the system* section.

Configuring the AWS image registry

The first difference is that we need to change the image to the full registry, so the cluster uses the images available in the ECS registry.

Remember, you need to specify the registry inside AWS so the AWS cluster can properly access it.

For example, in the `frontend/deployment.yaml` file, we need to define them in this way:

```
containers:
- name: frontend-service
  image: XXX.dkr.ecr.us-west-2.amazonaws.com/frontend:latest
  imagePullPolicy: Always
```

The image should pull from the AWS registry. The pull policy should be changed to force pulling from the cluster.

You can deploy in the remote server by applying the file, after creating the `example` namespace:

```
$ kubectl create namespace example
namespace/example created
$ kubectl apply -f frontend/deployment.yaml
deployment.apps/frontend created
```

After a bit, the deployment creates the pods:

```
$ kubectl get pods -n example
NAME                        READY STATUS  RESTARTS AGE
frontend-58898587d9-4hj8q 1/1   Running 0        13s
```

Now we need to change the rest of the elements. All the deployments need to be adapted to include the proper registry.

Check the code on GitHub to check all the `deployment.yaml` files.

Configuring the usage of an externally accessible load balancer

The second difference is to make the frontend service available externally, so internet traffic can access the cluster.

This is easily done by changing the service from `NodePort` to `LoadBalancer`. Check the `frontend/service.yaml` file:

```
apiVersion: v1
kind: Service
metadata:
    namespace: example
    labels:
        app: frontend-service
    name: frontend-service
```

```
spec:
    ports:
        - name: frontend
          port: 80
          targetPort: 8000
    selector:
        app: frontend
    type: LoadBalancer
```

This creates a new **Elastic Load Balancer** (**ELB**) that can be externally accessed. Now, let's start deploying.

Deploying the system

The whole system can be deployed, from the `Chapter07` subdirectory, with the following code:

```
$ kubectl apply --recursive -f .
deployment.apps/frontend unchanged
ingress.extensions/frontend created
service/frontend-service created
deployment.apps/thoughts-backend created
ingress.extensions/thoughts-backend-ingress created
service/thoughts-service created
deployment.apps/users-backend created
ingress.extensions/users-backend-ingress created
service/users-service created
```

This commands go iteratively through the subdirectories and applies any `.yaml` file.

After a few minutes, you should see everything up and running:

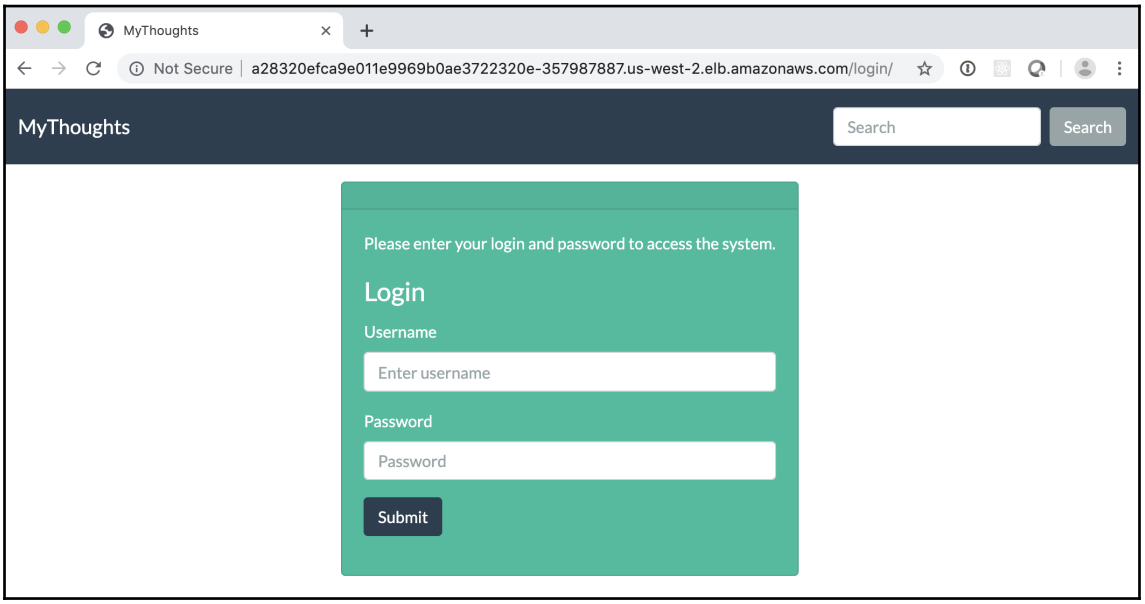
```
$ kubectl get pods -n example
NAME                               READY STATUS  RESTARTS AGE
frontend-58898587d9-dqc97          1/1   Running 0        3m
thoughts-backend-79f5594448-6vpf4 2/2   Running 0        3m
users-backend-794ff46b8-s424k      2/2   Running 0        3m
```

To obtain the public access point, you need to check the services:

```
$ kubectl get svc -n example
NAME              TYPE          CLUSTER-IP EXTERNAL-IP AGE
frontend-service LoadBalancer 10.100.152.177
a28320efca9e011e9969b0ae3722320e-357987887.us-west-2.elb.amazonaws.com 3m
thoughts-service NodePort 10.100.52.188 <none> 3m
users-service    NodePort 10.100.174.60 <none> 3m
```

Note that the frontend service has an external ELB DNS available.

If you put that DNS in a browser, you can access the service as follows:



Congratulations, you have your own cloud Kubernetes service. The DNS name the service is accessible under is not great, so we will see how to add a registered DNS name and expose it under an HTTPS endpoint.

Using HTTPS and TLS to secure external access

To provide a good service to your customers, your external endpoint should be served through HTTPS. This means that the communication between you and your customers is private, and it can't be sniffed throughout the network route.

The way HTTPS works is that the server and client encrypt the communication. To be sure that the server is who they say they are, there needs to be an SSL certificate issued by an authority that grants that the DNS is verified.

Remember, the point of HTTPS is *not* that the server is inherently trustworthy, but that the communication is private between the client and the server. The server can still be malicious. That's why verifying that a particular DNS does not contain misspellings is important.

You can get more information on how HTTPS works in this fantastic comic: `https://howhttps.works/`.

Obtaining a certificate for your external endpoint requires two stages:

- You own a particular DNS name, normally by buying it from a name registrar.
- You obtain a unique certificate for the DNS name by a recognized **Certificate Authority** (**CA**). The CA has to validate that you control the DNS name.

To help in promoting the usage of HTTPS, the non-profit *Let's Encrypt* (`https://letsencrypt.org`) supplies free certificates valid for 60 days. This will be more work than obtaining one through your cloud provider, but could be an option if money is tight.

These days, this process is very easy to do with cloud providers as they can act as both, simplifying the process.

The important element that needs to communicate through HTTPS is the edge of our network. The internal network where our own microservices are communicating doesn't require to be HTTPS, and HTTP will suffice. It needs to be a private network out of public interference, though.

Following our example, AWS allows us to create and associate a certificate with an ELB, serving traffic in HTTP.

Having AWS to serve HTTPS traffic ensures that we are using the latest and safest security protocols, such as **Transport Layer Security** (**TLS**) v1.3 (the latest at the time of writing), but also that it keeps backward compatibility with older protocols, such as SSL.

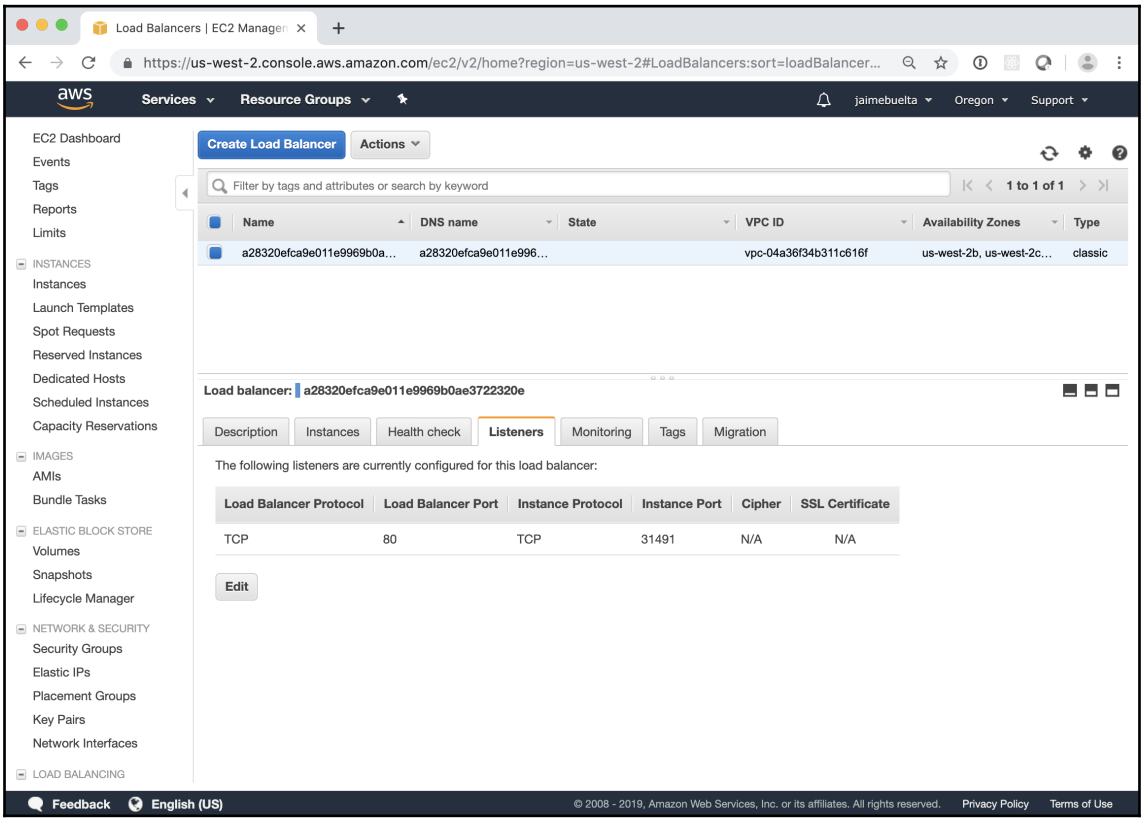
In other words, it is the best option to use the most secure environment by default.

The first step of setting HTTPS is either to buy a DNS domain name directly from AWS or to transfer the control to AWS. This can be done through their service Route 53. You can check the documentation at `https://aws.amazon.com/route53/`.

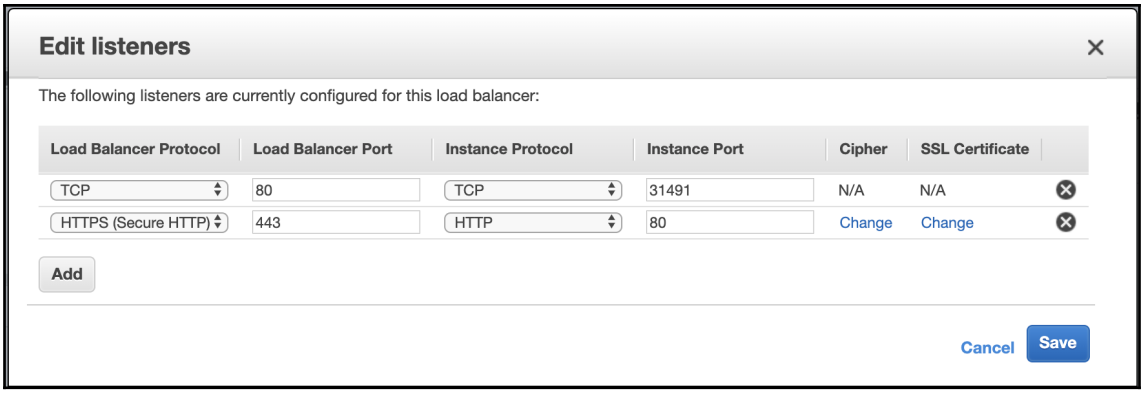
It is not strictly required to transfer your DNS to Amazon, as long as you can point it toward the externally facing ELB, but it helps with the integration and obtaining of certificates. You'll need to prove that you own the DNS record when creating a certificate, and using AWS makes it simple as they create a certificate to a DNS record they control. Check the documentation at `https://docs.aws.amazon.com/acm/latest/userguide/gs-acm-validate-dns.html`.

To enable HTTPS support on your ELB, let's check the following steps:

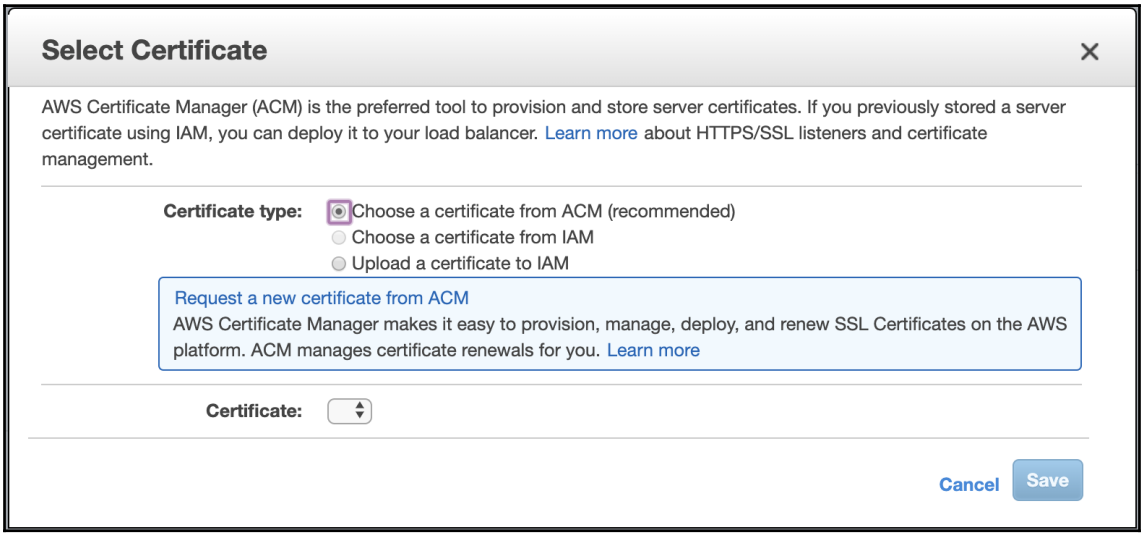
1. Go to **Listeners** in the AWS console:



2. Click on **Edit** and add a new rule for HTTPS support:



3. As you can see, it will require an SSL certificate. Click on **Change** to go to management:



4. From here, you can either add an existing certificate or buy one from Amazon.

Be sure to check the documentation about the load balancer in Amazon. There are several kinds of ELBs that can be used, and some have different features than others depending on your use case. For example, some of the new ELBs are able to redirect toward HTTPS even if your customer requests the data in HTTP. See the documentation at `https://aws.amazon.com/elasticloadbalancing/`.

Congratulations, now your external endpoint supports HTTPS, ensuring that your communications with your customers are private.

Being ready for migration to microservices

To operate smoothly while making the migration, you need to deploy a load balancer that allows you to quickly swap between backends and keeps the service up.

As we discussed in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, HAProxy is an excellent choice because it is very versatile and has a good UI that allows you to make operations quickly just by clicking on a web page. It also has an excellent stats page that allows you to monitor the status of the service.

AWS has an alternative to HAProxy called **Application Load Balancer** (**ALB**). This works as a feature-rich update on the ELB, which allows you to route different HTTP paths into different backend services.

HAProxy has a richer set of features and a better dashboard to interact with it. It can also be changed through a configuration file, which helps in controlling changes, as we will see in `Chapter 8`, *Using GitOps Principles*.

It is, obviously, only available if all the services are available in AWS, but it can be a good solution in that case, as it will be simpler and more aligned with the rest of the technical stack. Take a look at the documentation at `https://aws.amazon.com/blogs/aws/new-aws-application-load-balancer/`.

To deploy a load balancer in front of your service, I recommend not deploying it on Kubernetes, but running it in the same way as your traditional services. This kind of load balancer will be a critical part of the system, and removing uncertainty is important for a successful run. It's also a relatively simple service.

Keep in mind that a load balancer needs to be properly replicated, or it becomes a single point of failure. Amazon and other cloud providers allow you to set up an ELB or other kinds of load balancer toward your own deployment of load balancers, enabling the traffic to be balanced among them.

As an example, we've created an example configuration and the `docker-compose` file to quickly run it, but the configuration can be set up in whatever way your team is most comfortable with.

Running the example

The code is available on GitHub (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter07/haproxy`). We inherit from the HAProxy Docker image in Docker Hub (`https://hub.docker.com/_/haproxy/`), adding our own config file.

Let's take a look at the main elements in the config file, `haproxy.cfg`:

```
frontend haproxynode
    bind *:80
    mode http
    default_backend backendnodes

backend backendnodes
    balance roundrobin
    option forwardfor
    server aws a28320efca9e011e9969b0ae3722320e-357987887
              .us-west-2.elb.amazonaws.com:80 check
    server example www.example.com:80 check

listen stats
    bind *:8001
    stats enable
    stats uri /
    stats admin if TRUE
```

We define a frontend that accepts any requests into port `80` and sends the requests toward the backend. The backend balances the requests to two servers, `example` and `aws`. Basically, `example` points to `www.example.com` (a placeholder for your old service) and `aws` to the previously created load balancer.

We enable the stats server in port `8001` and allow admin access.

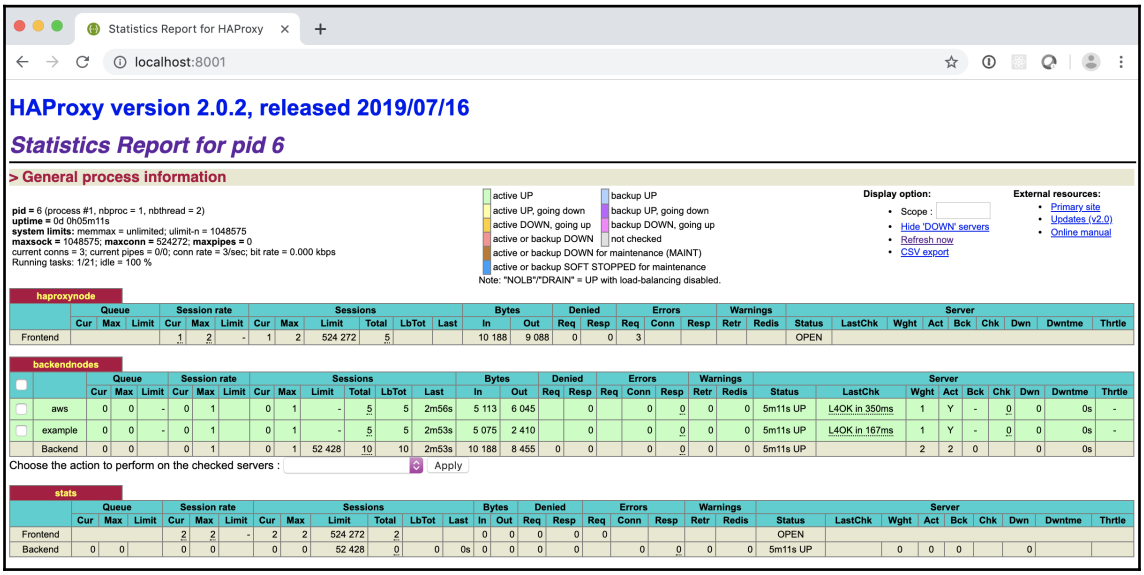
The `docker-compose` config starts the server and forwards the localhost ports towards the container ports `8000` (load balancer) and `8001` (stats). Start it with the following command:

```
$ docker-compose up --build proxy
...
```

Now we can access `localhost:8000`, which will alternate between the `thoughts` service and a 404 error.

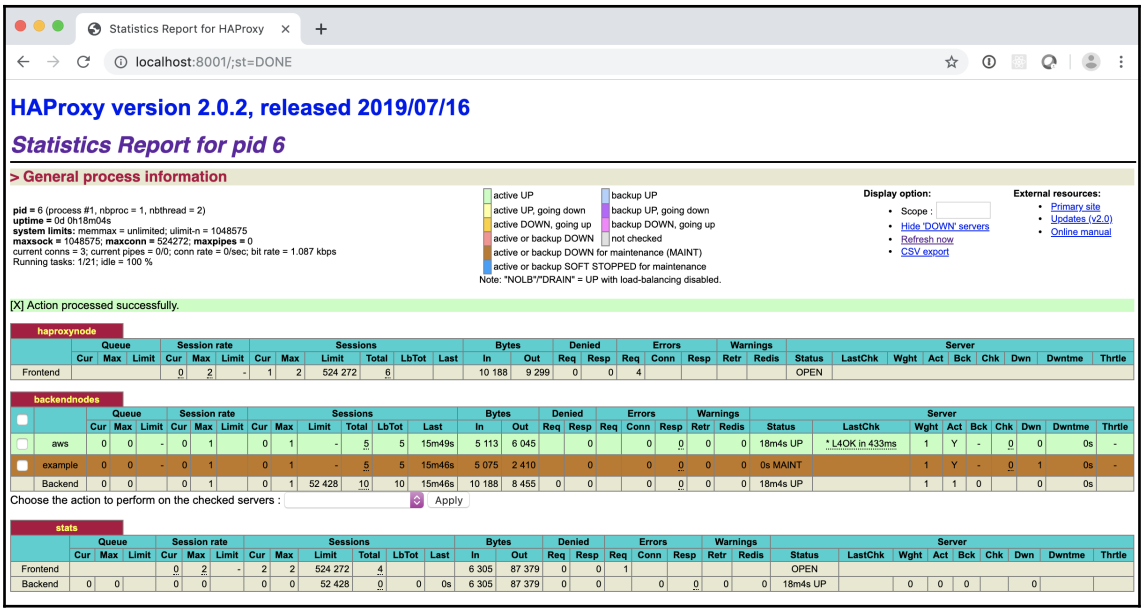
When calling `example.com` this way, we are forwarding the host request. This means we send a request requesting `Host:localhost` to `example.com`, which returns a 404 error. Be sure to check on your service that the same host information is accepted by all the backends.

Open the stats page to check the setup:



Check the entries for `aws` and `example` in the backend nodes. There is also a lot of interesting information, such as the number of requests, the last connection, data, and so on.

You can perform actions when checking the `example` backend, and then set state to **MAINT** in the drop-down menu. Once applied, the `example` backend is in maintenance mode and removed from the load balancer. The stats page is as follows:



Accessing the load balancer in `localhost:8000` now will only return the **thoughts** frontend. You can re-enable the backend setting it to the **READY** state.

There's a state called **DRAIN** that will stop new sessions going to the selected server, but existing sessions will keep going. This may be interesting in some configurations, but if the backend is truly stateless, moving directly to the **MAINT** state should be enough.

HAProxy can also be configured to use checks to ensure that the backend is available. We added in the example a commented one, which sends an HTTP command to check a return:

```
option httpchk HEAD / HTTP/1.1\r\nHost:\ example.com
```

The check will be the same to both backends, so it needs to be returned successfully. By default, it will be run every couple of seconds.

You can check the full HAProxy documentation at `http://www.haproxy.org/`. There are a lot of details that can be configured. Follow up with your team to be sure that the configuration of areas like timeouts, forwarding headers, and so on are correct.

The concept of the health check is also used in Kubernetes to ensure that pods and containers are ready to accept requests and are stable. We'll see how to ensure that a new image is deployed correctly in the next section.

Deploying a new Docker image smoothly

When deploying a service in a production environment, it is critically important to ensure that it is going to work smoothly to avoid interrupting the service.

Kubernetes and HAProxy are able to detect when a service is running correctly, and take action when it's not, but we need to give an endpoint that acts as a health check and configure it to be pinged at regular intervals, for early detection of problems.

For simplicity, we will use the root URL as a health check, but we can design specific endpoints to be tested. A good health checkup checks that the service is working as expected, but is light and quick. Avoid the temptation of over testing or performing an external verification that could make the endpoint take a long time.

An API endpoint that returns an empty response is a great example, as it checks that the whole piping system works, but it's very fast to answer.

In Kubernetes, there are two tests to ensure that a pod is working correctly, the readiness probe and the liveness probe.

The liveness probe

The liveness probe checks that a container is working properly. It is a process started in the container that returns correctly. If it returns an error (or more, depending on the config), Kubernetes will kill the container and restart it.

The liveness probe will be executed inside the container, so it needs to be valid. For a web service, adding a `curl` command is a good idea:

```
spec:
  containers:
  - name: frontend-service
    livenessProbe:
      exec:
        command:
        - curl
```

```
      - http://localhost:8000/
      initialDelaySeconds: 5
      periodSeconds: 30
```

While there are options such as checking if a TCP port is open or sending an HTTP request, running a command is the most versatile option. It can also be checked for debugging purposes. See the documentation for more options.

Be careful of being very aggressive on liveness probes. Each check puts some load on the container, so depending on load multiple probes can end up killing more containers than they should.

If your services are restarted often by the liveness probe, either the probe is too aggressive or the load is high for the number of containers, or a combination of both.

The probe is configured to wait for five seconds, and then run once every 30 seconds. By default, after three failed checks, it will restart the container.

The readiness probe

The readiness probe checks if the container is ready to accept more requests. It's a less aggressive version. If the tests return an error or timeout, the container won't be restarted, but it will be just labeled as unavailable.

The readiness probe is normally used to avoid accepting requests too early, but it will run after startup. A smart readiness probe can label when a container is at full capacity and can't accept more requests, but normally a probe configured in a similar way to the liveness prove will be enough.

The readiness probe is defined in the deployment configuration, in the same fashion as the liveness probe. Let's take a look:

```
spec:
  containers:
  - name: frontend-service
    readinessProbe:
      exec:
        command:
        - curl
        - http://localhost:8000/
        initialDelaySeconds: 5
        periodSeconds: 10
```

The readiness probe should be more aggressive than the liveness probe, as the result is safer. That's why `periodSeconds` is shorter. You may require both or not, depending on your particular use case, but readiness probes are required to enable rolling updates, as we'll see next.

The `frontend/deployment.yaml` deployment in the example code includes both probes. Check the Kubernetes documentation (`https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/`) for more details and options.

Be aware that the two probes are used for different objectives. The readiness probe delays the input of requests until the pod is ready, while the liveness probe helps with stuck containers.

A delay in the liveness probe getting back will restart the pod, so an increase in load could produce a cascade effect of restarting pods. Adjust accordingly, and remember that both probes don't need to repeat the same command.

Both readiness and liveness probes help Kubernetes to control how pods are created, which affects the update of a deployment.

Rolling updates

By default, each time that we update an image for deployment, the Kubernetes deployment will recreate the containers.

Notifying Kubernetes that a new version is available is not enough to push a new image to the registry, even if the tag is the same. You'll need to change the tag described in the `image` field in the deployment `.yaml` file.

We need to control how the images are being changed. To not interrupt the service, we need to perform a rolling update. This kind of update adds new containers, waits until they are ready, adds them to the pool, and removes the old containers. This deployment is a bit slower than removing all containers and restarting them, but it allows the service to be uninterrupted.

How this process is performed can be configured by tweaking the `strategy` section in the deployment:

```
spec:
    replicas: 4
    strategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 25%
        maxSurge: 1
```

Let's understand this code:

- `strategy` and `type` can be either `RollingUpdate` (the default) or `Recreate`, which stops existing pods and creates new ones.
- `maxUnavailable` defines the maximum number of unavailable pods during a change. This defines how quick o new containers will be added and old ones removed. It can be described as a percentage, like our example, or a fixed number.
- `maxSurge` defines the number of extra pods that can be created over the limit of desired pods. This can be a specific number or a percentage of the total.
- As we set `replicas` to `4`, in both cases the result is one pod. This means that during a change, up to one pod may be unavailable and that we will create the new pods one by one.

Higher numbers will perform the update faster but will consume more resources (`maxSurge`) or reduce more aggressively the available resources during the update (`maxUnavailable`).

For a small number of replicas, be conservative and grow the numbers when you are more comfortable with the process and have more resources.

Initially, scaling the pods manually will be easiest and best option. If the traffic is highly variable, with high peaks and low valleys, it may be worth autoscaling the cluster.

Autoscaling the cluster

We've seen before how to change the number of pods for a service, and how to add and remove nodes. This can be automated to describe some rules, allowing the cluster to change its resources elastically.

Keep in mind that autoscaling requires tweaking to adjust to your specific use case. This is a technique to use if the resource utilization changes greatly over time; for example, if there's a daily pattern where some hours present way more activity than others, or if there's a viral element that means the service multiplies the requests by 10 unexpectedly.

If your usage of servers is small and the utilization stays relatively constant, there's probably no need to add autoscaling.

The cluster can be scaled automatically up or down on two different fronts:

- The number of pods can be set to increase or decrease automatically in a Kubernetes configuration.
- The number of nodes can be set to increase or decrease automatically in AWS.

Both the number of pods and the number of nodes need to be in line with each other to allow natural growth.

If the number of pods increases without adding more hardware (nodes), the Kubernetes cluster won't have much more capacity, only the same resources allocated in a different distribution.

If the number of nodes increases without more pods created, at some point the extra nodes won't have pods to allocate, producing underutilization of resources. On the other hand, any new node added will have a cost associated, so we want to be properly using it.

To be able to automatically scale a pod, be sure that it is scalable. To ensure the pod is scalable check that it is an stateless web service and obtain all the information from an external source.

Note that, in our code example, the frontend pod is scalable, while the Thoughts and Users Backend is not, as they include their own database container the application connects to.

Creating a new pod creates a new empty database, which is not the expected behavior. This has been done on purpose to simplify the example code. The intended production deployment is, as described before, to connect to an external database instead.

Both Kubernetes configuration and EKS have features that allow changing the number of pods and nodes based on rules.

Creating a Kubernetes Horizontal Pod Autoscaler

In Kubernetes nomenclature, the service to scale pods up and down is called a **Horizontal Pod Autoscaler** (**HPA**).

This is because it requires a way of checking the measurement to scale. To enable these metrics, we need to deploy the Kubernetes metric server.

Deploying the Kubernetes metrics server

The Kubernetes metrics server captures internal low-level metrics such as CPU usage, memory, and so on. The HPA will capture these metrics and use them to scale the resources.

The Kubernetes metrics server is not the only available server for feeding metrics to the HPA, and other metrics systems can be defined. The list of the currently available adaptors is available in the Kubernetes metrics project (`https://github.com/kubernetes/metrics/blob/master/IMPLEMENTATIONS.md#custom-metrics-api`).

This allows for custom metrics to be defined as a target. Start first with default ones, though, and only move to custom ones if there are real limitations for your specific deployment.

To deploy the Kubernetes metrics server, download the latest version from the official project page (`https://github.com/kubernetes-incubator/metrics-server/releases`). At the time of writing, it was `0.3.3`.

Download the `tar.gz` file, which at the time of writing was `metrics-server-0.3.3.tar.gz`. Uncompress it and apply the version to the cluster:

```
$ tar -xzf metrics-server-0.3.3.tar.gz
$ cd metrics-server-0.3.3/deploy/1.8+/
$ kubectl apply -f .
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader
created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-
delegator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
serviceaccount/metrics-server created
deployment.extensions/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
```

You will see the new pod in the `kube-system` namespace:

```
$ kubectl get pods -n kube-system
NAME                              READY STATUS  RESTARTS AGE
...
metrics-server-56ff868bbf-cchzp 1/1   Running 0        42s
```

You can use the `kubectl top` command to get basic information about the nodes and pods:

```
$ kubectl top node
NAME                     CPU(cores) CPU% MEM(bytes) MEMORY%
ip-X.us-west-2.internal 57m         2%   547Mi       7%
ip-Y.us-west-2.internal 44m         2%   534Mi       7%
$ kubectl top pods -n example
$ kubectl top pods -n example
NAME                                CPU(cores) MEMORY(bytes)
frontend-5474c7c4ff-d4v77           2m         51Mi
frontend-5474c7c4ff-dlq6t           1m         50Mi
frontend-5474c7c4ff-km2sj           1m         51Mi
frontend-5474c7c4ff-rlvcc           2m         51Mi
thoughts-backend-79f5594448-cvdvm 1m           54Mi
users-backend-794ff46b8-m2c6w       1m         54Mi
```

To properly control what the limit of usage is, we need to configure in the deployment what is allocated and limit resources for it.

Configuring the resources in deployments

In the configuration for a container, we can specify what the requested resources are and the maximum resources for them.

They both inform Kubernetes about the expected memory and CPU usage for a container. When creating a new container, Kubernetes will automatically deploy it on a node that has enough resources to cover it.

In the `frontend/deployment.yaml` file, we include the following `resources` instances:

```
spec:
    containers:
    - name: frontend-service
      image: 033870383707.dkr.ecr.us-west-2
                 .amazonaws.com/frontend:latest
      imagePullPolicy: Always
      ...
      resources:
          requests:
              memory: "64M"
              cpu: "60m"
          limits:
              memory: "128M"
              cpu: "70m"
```

The initially requested memory is 64 MB, and 0.06 of a CPU core.

The resources for memory can also use Mi to the power of 2, which is equivalent to a megabyte (1000^2 bytes), which is called a mebibyte (2^{20} bytes). The difference is small in any case. You can use also G or T for bigger amounts.

The CPU resources are measured fractionally where 1 being a core in whatever system the node is running (for example, AWS vCPU). Note that 1000m, meaning 1000 milli CPUs is equivalent to a whole core.

The limits are 128 MB and 0.07 of a CPU core. The container won't be able to use more memory or CPU than the limit.

Aim at round simple numbers to understand the limits and requested resources. Don't expect to have them perfect the first time; the applications will change their consumption.

Measuring the metrics in an aggregated way, as we will talk about in `Chapter 11`, *Handling Change, Dependencies, and Secrets in the System*, will help you to see the evolution of the system and tweak it accordingly.

The limit creates the benchmark for the autoscaler, as it will be measured in a percentage of the resource.

Creating an HPA

To create a new HPA, we can use the `kubectl autoscale` command:

```
$ kubectl autoscale deployment frontend --cpu-percent=10 --min=2 --max=8 -n
example
horizontalpodautoscaler.autoscaling/frontend autoscaled
```

This creates a new HPA that targets the `frontend` deployment in the `example` namespace, and sets the number of pods to be between `2` and `8`. The parameter to scale is the CPU, which we set to 10% of the available CPU, averaged across all the pods. If it's higher, it will create new pods, and if it's lower, it will reduce them.

The 10% limit is used to be able to trigger the autoscaler and to demonstrate it.

The autoscaler works as a special kind of Kubernetes object, and it can be queried as such:

```
$ kubectl get hpa -n example
NAME     REFERENCE            TARGETS  MIN MAX REPLICAS AGE
frontend Deployment/frontend 2%/10%   2   8   4        80s
```

Note how the target says it is currently at around 2%, near the limit. This was designed with the small available CPU that will have a relatively high baseline.

After some minutes, the number of replicas will go down until the minimum is reached, `2`.

The downscaling may take a few minutes. This generally is the expected behavior, upscaling being more aggressive than downscaling.

To create some load, let's use the application Apache Bench (`ab`) in combination with a specially created endpoint in the frontend that uses a lot of CPU:

```
$ ab -n 100 http://<LOADBALANCER>.elb.amazonaws.com/load
Benchmarking <LOADBALANCER>.elb.amazonaws.com (be patient).....
```

Note that `ab` is a convenient test application that produces HTTP requests concurrently. If you prefer, you can hit the URL from a browser multiple times in quick succession.

Remember to add the load balancer DNS, as retrieved in the *Creating the cluster* section.

This will generate an extra CPU load in the cluster and will make the deployment scale up:

```
NAME     REFERENCE            TARGETS MIN MAX REPLICAS AGE
frontend Deployment/frontend 47%/10% 2   8   8        15m
```

After the requests are completed, and after some minutes, the number of pods will slowly scale down until hitting the two pods again.

But we need a way of scaling the nodes as well, or we won't be able to grow the total number of resources in the system.

Scaling the number of nodes in the cluster

The number of AWS instances that work as nodes in the EKS cluster can also be increased. This adds extra resources to the cluster and makes it possible to start more pods.

The underlying AWS service that allows that is an Auto Scaling group. This is a group of EC2 instances that share the same image and have a defined size, both for the minimum and maximum instances.

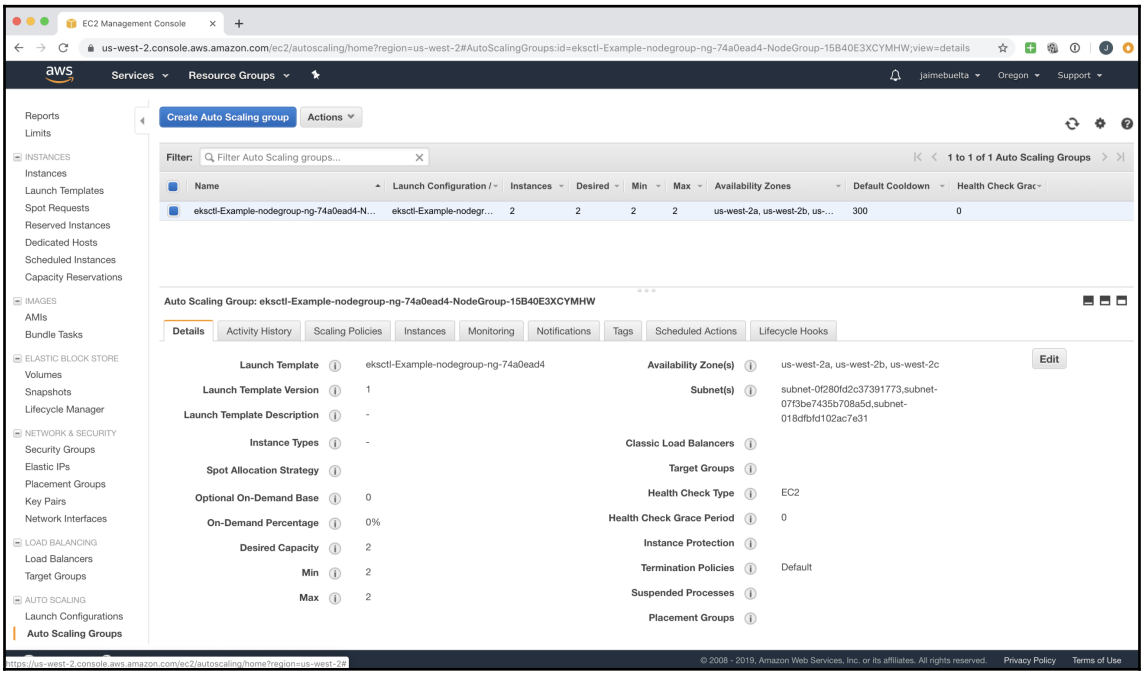
At the core of any EKS cluster, there's an Auto Scaling group that controls the nodes of the cluster. Note that `eksctl` creates and exposes the Auto Scaling group as a nodegroup:

```
$ eksctl get nodegroup --cluster Example
CLUSTER NODEGROUP    MIN  MAX  DESIRED INSTANCE IMAGE ID
Example ng-74a0ead4 2    2    2       m5.large ami-X
```

With `eksctl`, we can scale the cluster up or down manually as we described when creating the cluster:

```
$ eksctl scale nodegroup --cluster Example --name ng-74a0ead4 --nodes 4
[i] scaling nodegroup stack "eksctl-Example-nodegroup-ng-74a0ead4" in
cluster eksctl-Example-cluster
[i] scaling nodegroup, desired capacity from to 4, max size from 2 to 4
```

This nodegroup is also visible in the AWS console, under **EC2** | **Auto Scaling Groups**:



In the web interface, we have some interesting information that we can use to collect information about the Auto Scaling group. The **Activity History** tab allows you to see any scaling up or down event, and the **Monitoring** tab allows you to check metrics.

Most of the handling has been created automatically with `eksctl`, such as the **Instance Type** and the AMI-ID (the initial software on the instance, containing the operating system). They should be mainly controlled by `eksctl`.

If you need to change the **Instance Type**, `eksctl` requires you to create a new nodegroup, move all the pods, and then delete the old. You can learn more about the process in the `eksctl` documentation (`https://eksctl.io/usage/managing-nodegroups/`).

But from the web interface, it is easy to edit the scale parameters and to add policies for autoscaling.

Changing the parameters through the web interface may confuse the data retrieved in `eksctl`, as it's independently set up.

It is possible to install a Kubernetes autoscaler for AWS, but it requires a `secrets` configuration file to include a proper AMI in the autoscaler pod, with AWS permissions to add instances.

Describing the autoscale policy in AWS terms in code can also be quite confusing. The web interface makes it a bit easier. The pro is that you can describe everything in config files that can be under source control.

We will go with the web interface configuration, here, but you can follow the instructions at `https://eksctl.io/usage/autoscaling/`.

For scaling policies, there are two main components that can be created:

- **Scheduled actions**: They are scale up and down events that happen at defined times. The action can change the number of nodes through a combination of the desired number and the minimum and maximum number, for example, increasing the cluster during the weekend. The actions can be repeated periodically, such as each day or each hour. The action can also have an ending time, which will revert the values to the ones previously defined. This can be used to give a boost for a few hours if we expect extra load in the system, or to reduce costs during night hours.

- **Scaling policies**: These are policies that look for demand at a particular time and scale up or down the instances, between the described numbers. There are three types of policies: target tracking, step scaling, and simple scaling. Target tracking is the simplest, as it monitors the target (typically CPU usage) and scales up and down to keep close to the number. The other two policies require you to generate alerts using the AWS CloudWatch metrics system, which is more powerful but also requires using CloudWatch and a more complex configuration.

The number of nodes can change not only going up but also down, which implies deleting nodes.

Deleting nodes

When deleting a node, the pods running need to move to another node. This is handled automatically by Kubernetes, and EKS will do the operation in a safe way.

This can also happen if a node is down for any reason, such as an unexpected hardware problem. As we've seen before, the cluster is created in multiple availability zones to minimize risks, but some nodes may have problems if there's a problem in an Amazon availability zone.

Kubernetes was designed for this kind of problem, so it's good at moving pods from one node to another in unforeseen circumstances.

Moving a pod from one node to another is done by destroying the pod and restarting it in the new node. As pods are controlled by deployments, they will keep the proper number of pods, as described by the replicas or autoscale values.

Remember that pods are inherently volatile and should be designed so they can be destroyed and recreated.

Upscaling can also result in existing pods moving to other nodes to better utilize resources, though this is less common. An increase in the number of nodes is normally done at the same time as growing the number of pods.

Controlling the number of nodes requires thinking about the strategy to follow to achieve the best result, depending on the requirements.

Designing a winning autoscaling strategy

As we've seen, both kinds of autoscaling, pods and nodes, need to be related to each other. Keeping the number of nodes down reduces costs but limits the available resources that could be used for growing the number of pods.

Always keep in mind that autoscaling is a big numbers game. Unless you have enough load variation to justify it, tweaking it will produce cost savings that are not comparable to the cost of developing and maintaining the process. Run a cost analysis of expected gains and maintenance costs.

Prioritize simplicity when dealing with changing the size of a cluster. Scaling down during nights and weekends could save a lot of money, and it's much easier to handle than generating a complex CPU algorithm to detect highs and lows.

Keep in mind that autoscaling is not the only way of reducing costs with cloud providers, and can be used combined with other strategies.

For example, in AWS, reserving EC2 instances for a year or more allows you to greatly reduce the bill. They can be used for the cluster baseline and combined with more expensive on-demand instances for autoscaling, which yields an extra reduction in costs: `https://aws.amazon.com/ec2/pricing/reserved-instances/`.

As a general rule, you should aim to have an extra bit of hardware available to allow scaling pods, as this is faster. This is allowed in cases where different pods are scaled at different rates. It is possible, depending on the application, that when the usage of one service goes up, another goes down, which will keep the utilization in similar numbers.

This is not the use case that comes to mind, but for example, scheduled tasks during the night may make use of available resources that at daytime are being used by external requests.

They can work in different services, balancing automatically as the load changes from one service to the other.

Once the headroom is reduced, start scaling nodes. Always leave a security margin to avoid getting stuck in a situation where the nodes are not scaling fast enough and no more pods can be started because of a lack of resources.

The pod autoscaler can try to create new pods, and if there are no resources available, they won't be started. In the same fashion, if a node is removed, any Pod that is not deleted may not start because of a lack of resources.

Remember that we describe to Kubernetes the requirements for a new pod in the `resources` section of the deployment. Be sure that the numbers there are indicative of the required ones for the pod.

To ensure that the pods are adequately distributed across different nodes, you can use the Kubernetes affinity and anti-affinity rules. These rules allow defining whether pods of a certain kind should run in the same node or not.

This is useful, for example, to make sure that all kinds of pods are evenly distributed across zones, or for ensuring that two services are always deployed in the same node to reduce latency.

You can learn more about affinity and how to configure in this blog post: `https://supergiant.io/blog/learn-how-to-assign-pods-to-nodes-in-kubernetes-using-nodeselector-and-affinity-features/`, and in the Kubernetes official configuration (`https://kubernetes.io/docs/concepts/configuration/assign-pod-node/`).

In general, Kubernetes and `eksctl` defaults work fine for most applications. Use this advice only for advanced configuration.

Summary

In this chapter, we've seen how to apply a Kubernetes cluster into a production environment and create a Kubernetes cluster in a cloud provider, in this case, AWS. We've seen how to set up our Docker registries, create a cluster using EKS, and adapt our existing YAML files so they are ready for the environment.

Remember that, though we used AWS as an example, all of the elements we discussed are available in other cloud providers. Check their documentation to see if they work better for you.

We also saw how to deploy an ELB so the cluster is available to the public interface, and how to enable HTTPS support on it.

We discussed the different elements of deployments to make the cluster more resilient and to deploy new versions smoothly, not interrupting the service—both by using HAProxy to be able to quickly enable or disable services and by making sure that changing the container image is done in an orderly fashion.

We also covered how autoscaling can help rationalize the use of resources and allow you to cover peaks in load in the system, both by creating more pods and by adding more AWS instances to add resources to the cluster when needed and remove them to avoid needless costs.

In the next chapter, we will see how to control the state of the Kubernetes cluster using GitOps principles to be sure that any changes on it are properly reviewed and captured.

Questions

1. What are the main disadvantages of managing your own Kubernetes cluster?
2. Can you name some commercial cloud providers that have a managed Kubernetes solution?
3. Is there any action you need to do to be able to push to an AWS Docker registry?
4. What tool do we use to set up an EKS cluster?
5. What are the main changes we did in this chapter to adapt the YAML files from previous chapters?
6. Are there any Kubernetes elements that are not required in the cluster from this chapter?
7. Why do we need to control the DNS associated with an SSL certificate?
8. What is the difference between the liveness and readiness probes?
9. Why are rolling updates important in production environments?
10. What is the difference between autoscaling pods and nodes?
11. In this chapter, we deployed our own database containers. In production this will change, as it's required to connect to an already existing external database. How would you change the configuration to do so?

Further reading

To learn more about how to use AWS, networking capabilities, which are vast, you can check out the book *AWS Networking Cookbook* (`https://www.packtpub.com/eu/virtualization-and-cloud/aws-networking-cookbook`). To learn how to ensure that you set up a secure system in AWS, read *AWS: Security Best Practices on AWS* (`https://www.packtpub.com/eu/virtualization-and-cloud/aws-security-best-practices-aws`).

8
Using GitOps Principles

After seeing how to configure a Kubernetes cluster, we will learn how to do it using GitOps practices instead of applying manual commands and files. GitOps means managing the cluster configuration using a Git repo to store and track the YAML files that contain the configuration. We will see how to link a GitHub repo with a cluster, so that it gets updated regularly, using Flux.

This method allows us to store the configuration in a deterministic way, describing the changes to infrastructure in code. The changes can be reviewed and the cluster can be recovered from scratch or duplicated, as we will see in `Chapter 9`, *Managing Workflows*.

The following topics will be covered in this chapter:

- Understanding the description of GitOps
- Setting up Flux to control the Kubernetes cluster
- Configuring GitHub
- Making a Kubernetes cluster change through GitHub
- Working in production

By the end of the chapter, you will know how to store the Kubernetes configuration in a Git repository and apply automatically any changes that are merged into the main branch.

Technical requirements

The code for the examples in the chapter is available on GitHub: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter08`.

You will need to install the `fluxctl` tool. We will then use this tool to sync manually and to obtain an SSH key to allow Flux to interact with the Git repo. See how to install it in its documentation: `https://docs.fluxcd.io/en/stable/tutorials/get-started.html`.

Understanding the description of GitOps

A big traditional problem in operations has been ensuring that the different servers maintain a proper configuration. When you have a fleet of servers, deploying a service and keeping them properly configured is not a straightforward task.

For this chapter, we will use *configuration* as a way of describing a service and all the required configuration to run it in production. This includes the particular version of the service, but also things such as the underlying infrastructure (OS version, number of servers, and so on) or packages and configuration of the dependent services (load balancers, third-party libraries, and so on).

Configuration management will, therefore, be the way to make changes to that.

Keeping configuration on track in all servers is challenging as the infrastructure grows. The most common change is to deploy a new version of a service, but there are other possibilities. For example, there's a new server being added that needs to be added to the load balancer, new configuration tweaks for NGINX to fix a security bug, or a new environment variable for the service to enable a feature.

The initial stage is manual configuration, but that gets difficult to do after a while.

Managing configuration

The manual configuration means that someone on the team keeps track of a small number of servers, and when a change is required, logs individually on each server and makes the required changes.

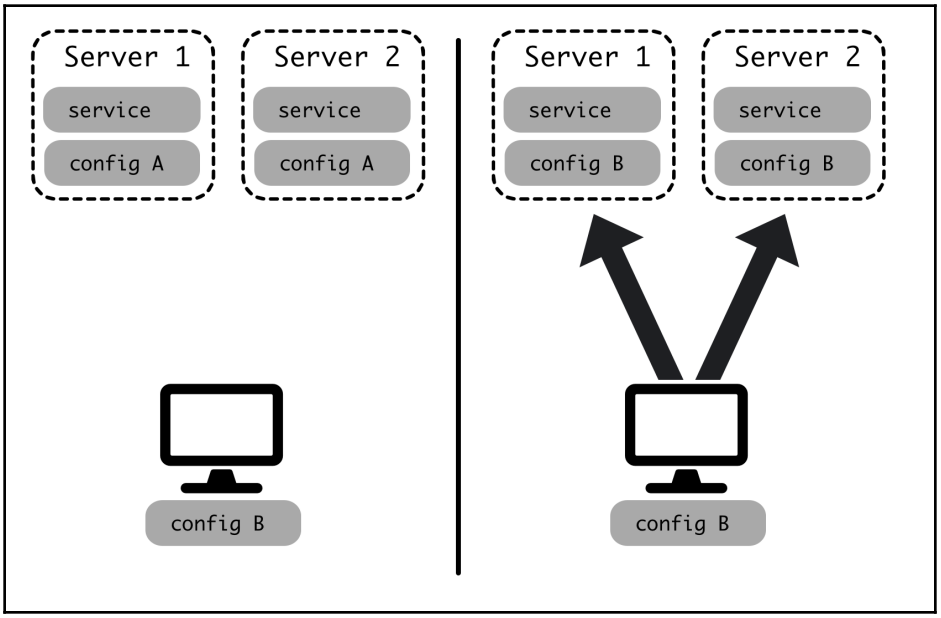
This way of operating is work-intensive and error prone with multiple servers, as they could easily diverge.

So, after some time, it can be improved through some scripts using Fabric (`http://www.fabfile.org/`) or Capistrano (`https://capistranorb.com/`). The basic model is to push the configuration and the new code to the servers and perform a number of automated tasks, restarting the service at the end. Typically, this is done directly from the computers of the team, as a manual step.

The code and configuration are normally present on Git, but the manual process makes it possible to change this, as it is detached. If you work this way, ensure that only files stored under source control are being deployed.

Some elements for server maintenance, like operating system upgrades or updating libraries, may still require to be done manually.

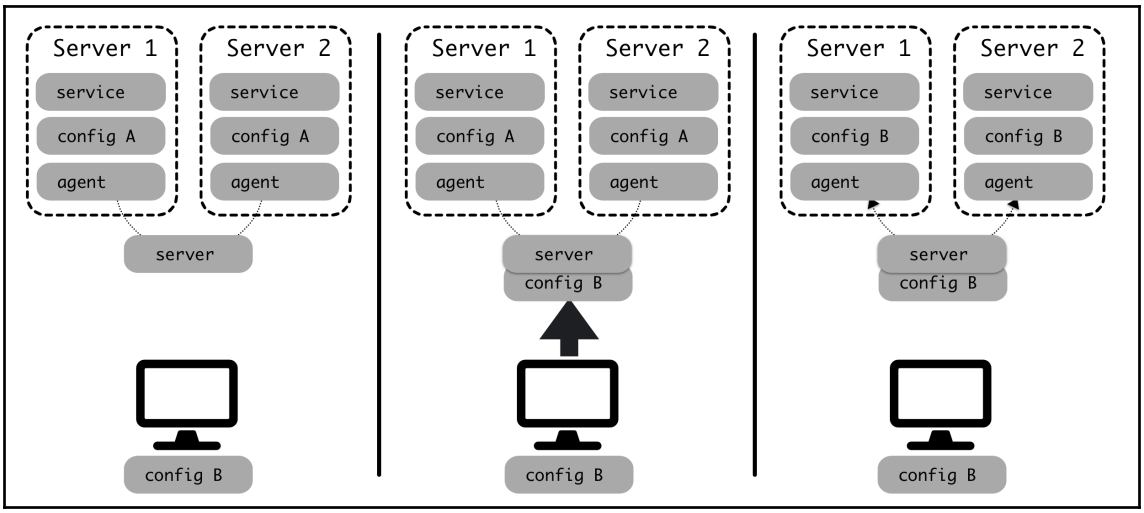
The following diagram shows how the code is pushed from the computer of the team member that makes the configuration change:



In this stage, new infrastructure can be added manually or by using a tool such as Terraform (`https://www.terraform.io/`) to interact with cloud services.

A more sophisticated alternative is to use tools such as Puppet (`https://puppet.com/`) or Chef (`https://www.chef.io/`). They work with a client-server architecture. They allow us to describe the state of the servers using their own declarative language, and when that's changed in the server, all clients will update to follow the definition. The server will report any problems or deviations and will centralize the configuration definition.

This process is summarized in the following diagram:



In some cases, these tools may be able to allocate resources in cloud services; for example, adding a new EC2 instance in AWS.

A configuration management tool also helps in monitoring and performs a number of remediation tasks. For example, it can restart services that should be running, or retry if there has been a problem changing the configuration.

It also scales better for a higher number of servers.

All these strategies require specialized tools and are normally handled by a specific operations team. This makes the configuration out of reach for developers requiring coordination between them in order to make a configuration update.

This division of work creates some friction and, over time, the DevOps movement has proposed other ways of structuring this work.

Understanding DevOps

The traditional way of dividing the work was to create an operations team that will control the infrastructure and deployments, and a development team that creates the service.

The problem with this approach is that developers normally won't understand truly how their code works in production, and, simultaneously, operations won't exactly know what a deployment contains. This can lead to situations of *I don't know what it is/I don't know where it is*, where there is a chasm between the two teams. DevOps was eventually created as an approach to fill that gap.

A typical problem is one where a service frequently fails in production and is detected by operations, which performs remediation tactics (for example, restarting the service).

However, the development team doesn't know exactly what makes it fail and they have other pressing priorities, so they won't fix the problem.

Over time, this may compromise the stability of the system.

DevOps is a set of techniques to improve collaboration between the operation side and the development side. It aims to allow quick deployment by making developers aware of the whole operation side, and to simplify operations by using automation as much as possible.

The core of it is to empower teams to allow them to control their own infrastructure and deployments, speeding up the rate of deployment and understanding the infrastructure to help identify problems early. The team should be autonomous in deploying and supporting the infrastructure.

To enable DevOps practices, you need some tools to control the different operations in a controlled way. GitOps is an interesting choice for that, especially if you use Kubernetes.

Defining GitOps

The idea of GitOps is simple—we use Git to describe our infrastructure and configuration management. Any change to a defined branch will trigger the relevant changes.

If you are able to define the whole system through code, Git gives you a lot of advantages:

- Any change to either infrastructure or configuration management is versioned. They are explicit and can be rolled back if they have problems. Changes between versions can be observed through diffs, which is a normal Git operation.

- A Git repo can act as a backup that can enable recovery from scratch if there's a catastrophic failure in the underlying hardware.
- It is the most common source control tool. Everyone in the company likely knows how it works and can use it. It also easily integrates with existing workflows, like reviews.

The GitOps concept was introduced and named by Weaveworks in a blog post (`https://www.weave.works/blog/gitops-operations-by-pull-request`). Since then, it has been used more and more in companies.

While GitOps could be applied to other kinds of deployments (and it certainly has been), it has good synergy with Kubernetes, which actually was the description in the Weaveworks blog post.

A Kubernetes cluster can be completely configured using YAML files, which encapsulates almost the whole definition of the system. As we saw in the previous chapter, this may include the definition of elements such as load balancers. The elements external to the Kubernetes cluster, such as an external DNS, which are not included in the YAML files, are rare to change.

> The servers and infrastructure can be automated with other tools, like Terraform, or with the automated procedures described in `Chapter 7`, *Configuring and Securing the Production System*.

> For pragmatic reasons, it is entirely feasible that some infrastructure operations are manual. For example, upgrading the Kubernetes version of an EKS cluster is an operation that can be done through the AWS console, and it is rare enough that it is fine to do so manually.
>
> It is also fine to have these kinds of operations remain manual, since automating them probably won't pay dividends.

As we saw in `Chapter 6`, *Local Development with Kubernetes*, the Kubernetes YAML files contain element definitions that can be applied with the `kubectl apply -f <file>` commands. Kubernetes is quite flexible since a single file can contain multiple elements or a single one.

Grouping all the YAML files under a directory structure and getting them under Git control makes a very explicit way of applying changes. This is the way that we will operate.

This operation is not complicated, but we will use an existing tool, created by Weaveworks, called **Flux**.

Setting up Flux to control the Kubernetes cluster

Flux (`https://github.com/fluxcd/flux`) is a tool that ensures that the state of a Kubernetes cluster matches the files stored in a Git repo.

It gets deployed inside the Kubernetes cluster, as another deployment. It runs every 5 minutes and checks with the Git repo and Docker registry. Then, it applies any changes. This helps with access to the Git repo, as there's no need to create any push mechanism inside a CI system.

We will see how to start a Flux container inside Kubernetes that pulls from the GitHub repo.

Starting the system

For simplicity, we will use the local Kubernetes. We will use the images described in `Chapter 6`, *Local Development with Kubernetes*, so be sure to run the following commands:

```
$ cd Chapter06
$ cd frontend
$ docker-compose build server
...
Successfully tagged thoughts_frontend:latest
$ cd ..
$ cd thoughts_backend/
$ docker-compose build server db
...
Successfully tagged thoughts_frontend:latest
$ cd ..
$ cd users_backend
$ docker-compose build server db
...
Successfully tagged users_server:latest
```

The basic Kubernetes configuration is stored in the example folder (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter08/example`) subdirectory.

You can deploy the entire system with the following commands:

```
$ cd Chapter08/example
$ kubectl apply -f namespace.yaml
namespace/example created
$ kubectl apply -f . --recursive
deployment.apps/frontend created
ingress.extensions/frontend-ingress created
service/frontend-service created
namespace/example unchanged
deployment.apps/thoughts-backend created
ingress.extensions/thoughts-backend-ingress created
service/thoughts-service created
deployment.apps/users-backend created
ingress.extensions/users-backend-ingress created
service/users-service created
```

This creates the entire system.

Apply the `namespace.yaml` file to avoid not being able to deploy elements as the namespace is not present, but you can run the `kubectl apply -f . --recursive` command twice.

If you check the system, it should be deployed now, as shown by running the `kubectl get pods` command:

```
$ kubectl get pods -n example
NAME                     READY STATUS  RESTARTS AGE
frontend-j75fp           1/1   Running 0        4m
frontend-n85fk           1/1   Running 0        4m
frontend-nqndl           1/1   Running 0        4m
frontend-xnljj           1/1   Running 0        4m
thoughts-backend-f7tq7 2/2   Running 0        4m
users-backend-7wzts    2/2   Running 0        4m
```

Note that there are four copies of `frontend`. We will change the number of pods during this chapter as an example of how to change a deployment.

Now, delete the deployment to start from scratch:

```
$ kubectl delete namespace example
namespace "example" deleted
```

For more details about this setup, check the *Deploying the full system locally* section in `Chapter 6`, *Local Development with Kubernetes*.

Configuring Flux

We will prepare a Flux system, which will help us keep track of our Git configuration. We prepared one based on the Flux example in this repo (`https://github.com/fluxcd/flux/tree/master/deploy`), and it's available in the `Chapter08/flux` subdirectory.

The main file is `flux-deployment.yaml`. Most of it is commented boilerplate, but take a look at the definition of the repo to pull from:

```
# Replace the following URL to change the Git repository used by Flux.
- --git-url=git@github.com:PacktPublishing/Hands-On-Docker-for-
Microservices-with-Python.git
- --git-branch=master
# Include this if you want to restrict the manifests considered by flux
# to those under the following relative paths in the git repository
- --git-path=Chapter08/example
```

These lines tell Flux the repo to use, the branch, and any path. If the path is commented, which it probably is in your case, it uses the whole repo. In the next section, we will need to change the repo to use your own one.

Note that we use the `flux` namespace to deploy all these elements. You can reuse your main namespace or use the default one if it works better for you.

To use Flux, create the namespace and then apply the full `flux` directory:

```
$ kubectl apply -f flux/namespace.yaml
namespace/flux created
$ kubectl apply -f flux/
serviceaccount/flux created
clusterrole.rbac.authorization.k8s.io/flux created
clusterrolebinding.rbac.authorization.k8s.io/flux created
deployment.apps/flux created
secret/flux-git-deploy created
deployment.apps/memcached created
service/memcached created
namespace/flux unchanged
```

Using the following code, you can check that everything is running as expected:

```
$ kubectl get pods -n flux
NAME                          READY STATUS  RESTARTS AGE
flux-75fff6bbf7-bfnq6         1/1   Running 0        34s
memcached-84f9f4d566-jv6gp 1/1   Running 0        34s
```

But, to be able to deploy from a Git repo, we need to configure it.

Configuring GitHub

Though we can configure any Git repo, typically, we will use GitHub to set it up. We will need to set up a valid key to access the Git repo.

The easiest way to do this is to allow Flux to generate its own key, and add it to the GitHub repo. But to be able to do so, we need to create our own GitHub repo.

Forking the GitHub repo

The first step in configuring the repo is to fork it. Let's look at the following steps for more details:

1. Go to the page of the GitHub repo for the code (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/`) and click in **Fork** on the top-right corner to generate your own copy.
2. Once you have your own copy, it will have a URL similar to the following:

   ```
   https://github.com/<YOUR GITHUB USER>/Hands-On-Docker-for-Microservices-with-Python/
   ```

3. Now, you need to replace it in the `Chapter08/flux/flux-deployment.yaml` file for the `--git-url` parameter.
4. Once you change it, reapply the Flux configuration with the following command:

   ```
   $ kubectl apply -f flux/flux-deployment.yaml
   deployment.apps/flux changed
   ```

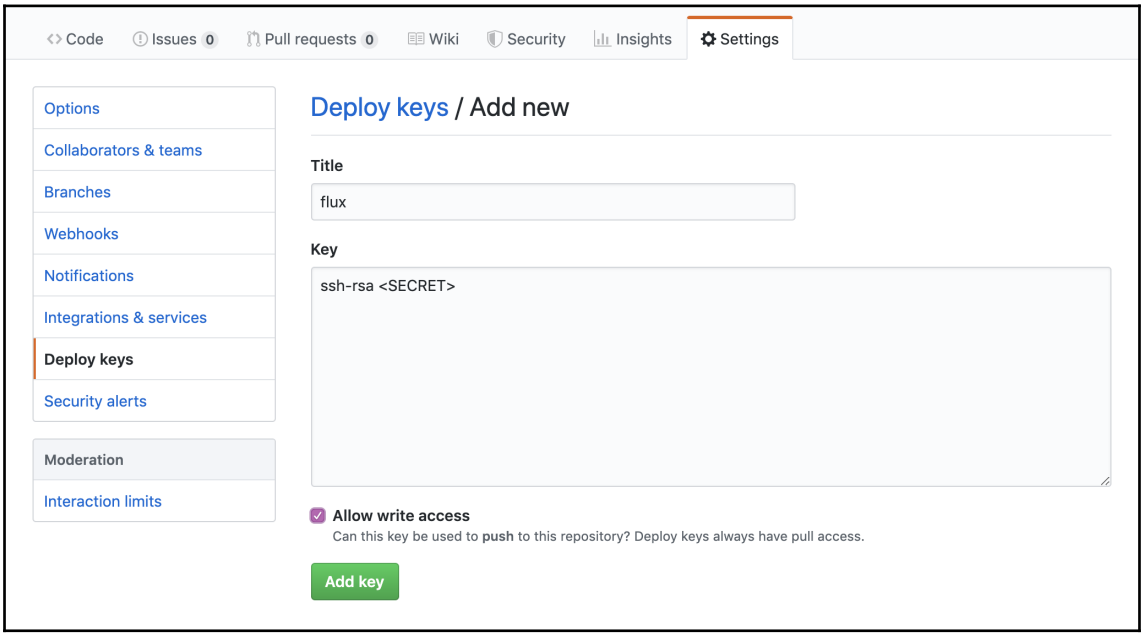
Now, Flux is tracking your own repo under your full control, and you can make changes to it. First of all, we need to allow Flux to access the GitHub repo, which can be achieved through a deploy key.

Adding a deploy key

To allow Flux to access GitHub, we need to add its secret key as a valid deploy key. Using `fluxctl`, it is easy to get the current `ssh` key; just run the following command:

```
$ fluxctl identity --k8s-fwd-ns flux
ssh-rsa <secret key>
```

With that information, go to the **Settings** | **Deploy keys** section on your forked GitHub project. Fill the title with a descriptive name, the **Key** section with your secret key as obtained before, and then select **Add key**:



Be sure to select the checkbox for **Allow write access**. Now, Flux will be able to contact GitHub.

The next step is to synchronize the state on GitHub and the cluster.

Syncing Flux

We can sync with Flux, so the description in GitHub is applied in the cluster, using the following command:

```
$ fluxctl sync --k8s-fwd-ns flux
Synchronizing with git@github.com:<repo>.git
Revision of master to apply is daf1b12
Waiting for daf1b12 to be applied ...
Done.
Macbook Pro:Chapter08 $ kubectl get pods -n example
NAME                    READY STATUS  RESTARTS AGE
frontend-8srpc          1/1   Running 0        24s
frontend-cfrvk          1/1   Running 0        24s
frontend-kk4hj          1/1   Running 0        24s
frontend-vq4vf          1/1   Running 0        24s
thoughts-backend-zz8jw 2/2   Running 0        24s
users-backend-jrvcr     2/2   Running 0        24s
```

It will take a bit to sync, and it's possible that you will get an error stating that it is cloning the repo:

```
$ fluxctl sync --k8s-fwd-ns flux
Error: git repository git@github.com:<repo>.git is not ready to sync
(status: cloned)
Run 'fluxctl sync --help' for usage
```

Wait for a couple of minutes and try again:

```
$ fluxctl sync --k8s-fwd-ns flux
Synchronizing with git@github.com:<repo>.git
Revision of master to apply is daf1b12
Waiting for daf1b12 to be applied ...
Done.
$
```

Your Flux deployment and, therefore, the local Kubernetes cluster are now in sync with the configuration in Git and will update with any change.

Making a Kubernetes cluster change through GitHub

Your local Kubernetes cluster, through Flux, will update to reflect changes in the Git repo. Any change in Git will be propagated to the cluster after a few minutes.

Let's see this with a test updating the number of pods in the frontend deployment:

1. Change the `Chapter08/example/frontend/deployment.yaml` file in your forked repo as described here:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: frontend
    labels:
        app: frontend
    namespace: example
spec:
    replicas: 2
```

   This changes the number of replicas from `4` to `2`.

2. Commit the change into the `master` branch and push into the GitHub repo.
3. Monitor the cluster with the following command:

```
$ kubectl get pods -n example -w
```

   You will see how the number of frontend pods will decrease after a few minutes. You can speed it up by manually syncing Flux.

4. Revert the change and see how they'll be added.

Flux won't delete elements to avoid problems. This means that removing a file of a deployment or service won't eliminate it from the repo. To do so, you need to remove it manually.

You can disable pods controlled by a deployment by setting the number of replicas to zero.

Congratulations! You now have a cluster controlled by a GitHub repo.

Let's look at some ideas on how to use this methodology efficiently in production environments.

Working in production

GitOps is mainly aimed at working on production environments, which are bigger and more complex than the example local cluster we used in this chapter. In this section, we will describe how to use the advantages of Git to improve clarity in terms of deployments and changes, and how to be sure that we structure the different files under source control to avoid confusion.

Creating structure

Structuring the YAML files is critical for a big deployment. Technically, you can join everything in a single file, but that's not the best way of handling it when it grows. Kubernetes allows a great deal of flexibility, so try to find a structure that works for you.

A simple one is to create subdirectories by namespace and then by microservice. This is the way we have structured it in this example. This structure keeps related elements together and has a clear path for anyone touching a microservice. If deployments affect only one microservice (as they should, as we discussed in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, in the *Parallel deployment and development speed* section), this keeps changes in the same subdirectory.

But don't feel limited to this structure. If it makes sense for you, you can try something different; for example, making a division by element, that is, all deployments under a directory, all services under another, and so on. Don't be afraid to experiment and move elements, searching for the best structure for your project.

All these files are under source control in GitHub, which allows us to use their features to our advantage.

Using GitHub features

Given that any merge to the main branch will trigger a change in the cluster, this should be reviewed before going live.

You can do it by requiring a pull request that needs approval prior to merging. The approval can come from an Ops team dedicated to keeping track of the cluster, or by the owner of the microservice; for example, a team lead or manager.

You can enforce code owners natively in GitHub. This means that a change in a particular file or directory requires some user or team to approve it. Check the GitHub documentation for more info (`https://help.github.com/en/articles/about-code-owners`).

A single GitHub repo can also keep track of more than one environment, for example, a staging environment to run tests, and a production environment that is available to customers. You can divide them either by branch or subdirectory.

But GitHub features are not the only ones available, the regular Git tag is extremely versatile and allows us to define specific containers to deploy.

Working with tags

In this example, we've worked with the `latest` tag for the images. This uses the most recently built container, which can change each time that an image is built. For production, we should use a specific tag linked to an immutable container, as we discussed in `Chapter 3`, *Build, Run, and Test Your Service Using Docker*, in the *Using a remote registry* section, and in `Chapter 4`, *Creating a Pipeline and Workflow*, in the *Pushing Docker Images from Travis CI* section.

This means replacing the following lines:

```
spec:
  containers:
  - name: frontend-service
    image: thoughts_frontend:latest
```

We replace them with the following:

```
spec:
  containers:
  - name: frontend-service
    image: <registry>/thoughts_frontend:v1.5
```

This is where being able to update the images in a controlled way shines. You will use the pipeline (as described in `Chapter 4`, *Creating a Pipeline and Workflow*) to build and push the tagged images to the remote registry, and then you can control which specific version is deployed in the cluster.

In some cases, it may be needed to stop the syncing. Flux works with the concept of workloads, which are updateable elements, in the same way as deployments.

You can stop the automatic updating of them or control how they are updated. Refer to the documentation for more information: `https://github.com/fluxcd/flux/blob/master/docs/using/fluxctl.md#workloads`.

Making this version under Git control makes it easy for developers revert to, or to come back to, a previous version.

To follow the continuous integration principles, try to make small changes and apply them quickly. Git will help you revert a bad change, but small incremental changes are easy to test and reduce the risk of breaking the system.

Most of the operations will be simple changes—either change the version of the image to be deployed, or tweak parameters, such as the number of replicas or an environment variable.

Summary

We started this chapter with a review of the most common different kinds of configuration management strategies, and how they tend to be applied as a project grows. We discussed how the DevOps approach makes teams take ownership of their deployments and helps to fill the traditional gap between development and operations.

We saw how the latest approach called GitOps works very well with a Kubernetes cluster, as the configuration is tightly described as a set of files. We went through the advantages of using Git to track the configuration.

We introduced Flux, a tool that gets deployed inside your cluster and pulls changes from a Git repo branch. We presented an example configuration, deployed it in our local Kubernetes cluster, and configured GitHub to be able to work with it. This enabled any push to our Git branch in GitHub to be reflected in the local cluster.

We ended the chapter with some strategies for working in production. We looked at making sure that the Kubernetes YAML files are properly structured, taking advantage of the GitHub features, and we learned how to release and roll back tagged images.

In the next chapter, we will describe the process of a full developing cycle for the cluster, from the introduction of a new feature until it gets deployed in a production environment. We will describe some useful strategies when working in live systems, so that the deployed code works smoothly and is of a high quality.

Questions

1. What is the difference between using a script to push new code to servers and using a configuration management tool such as Puppet?
2. What is the core idea behind DevOps?
3. What are the advantages of using GitOps?
4. Can GitOps only be used in a Kubernetes cluster?
5. Where is the Flux deployment located?
6. What do you need to configure in GitHub in order to allow Flux to access it?
7. For working in production environments, which features does GitHub provide that can help ensure control over deployments?

Further reading

You can learn more about DevOps practices and ideas in the following books: *Practical DevOps – Second Edition* (`https://www.packtpub.com/virtualization-and-cloud/practical-devops-second-edition`), and *DevOps Paradox* (`https://www.packtpub.com/web-development/devops-paradox`).

9
Managing Workflows

In this chapter, we will bring the different processes we described in the previous chapters together into a general workflow so that we can make changes to a single microservice. We will move on from the process of getting a new feature request to developing it locally, reviewing it, testing it in a demo environment, and approving the changes and releasing them to the live cluster.

This is related to the pipeline concept we introduced in `Chapter 4`, *Creating a Pipeline and Workflow*. In this chapter, however, we will talk about the processes of the task. The pipeline and build structures are there to ensure that any proposed change follows quality standards. In this chapter, we will focus on the teamwork aspects of the technology and how to allow smooth interaction while keeping track of the different changes.

In this chapter, we will cover the following topics:

- Understanding the life cycle of a feature
- Reviewing and approving a new feature
- Setting up multiple environments
- Scaling the workflow and making it work

By the end of this chapter, we will have a clear view of the different steps that are involved in setting up a new feature for one of our microservices, and how can we use multiple environments to test and ensure that the release is successful.

Understanding the life cycle of a feature

Following on from agile principles, the main objective of any team is to be able to implement new features quickly, without compromising the quality or stability of the system. The first element of change comes in the shape of a **feature request**.

A feature request is a description of a change in the system in non-technical terms. Feature requests are normally generated by non-engineers (product owners, managers, and CEOs) who are looking to improve the system for business-related reasons, such as making a better product or increasing revenue.

Feature requests can be simple, such as *updating the logo of the company in the main page*, or big and complicated, such as *adding support to the new 5G network*. Feature requests may include bug reports. While they don't usually, they will for the purpose of this chapter.

Complicated feature requests may need to be subdivided into smaller self-contained feature requests so that we can iterate in small increments.

Our focus is on the elements that need to be taken into account due to the microservices approaches and practices more than agile practices. Such practices deal with how to structure feature requests into tasks and estimations, but they are not specific to the underlying technologies.

Take a look at the *Further reading* section at the end of this chapter to find out more about agile practices and methodologies.

In a monolith, we have all the elements under the same code base. So, no matter how complicated a particular feature request is, only one system will be affected. There is only a single system in a monolith. However, this isn't the case once we migrate to microservices.

In a microservice architecture, we need to analyze any incoming feature request in terms of what microservice or microservices it affects. If we design our microservices properly, most of the requests will only affect a single microservice. However, eventually, some of the feature requests will be too big to fit neatly into a single microservice and will need to be divided into two or more steps, each changing a different microservice.

For example, if we have a new feature request that allows us to mention a user in the text of a thought (similar to how a mention works on Twitter), then this mention will have to be stored in the Thoughts Backend and be displayed in the Frontend. This feature affects two microservices: the Frontend and the Thoughts Backend.

In this section, we are referring to concepts that we introduced in the previous chapters and are joining them together from a global point of view.

In the next subsection, we will look at the features that affect multiple microservices.

Features that affect multiple microservices

For a multiple microservice feature request, you need to divide the feature into several technical features, with each one affecting a single microservice.

Each technical feature should cover an aspect that's relevant to the microservice it affects. If each microservice has a clear purpose and objective, the feature will be completed and generalized so that it can be used for later requests.

The basis for a successful microservice architecture is to have loosely coupled services. Ensuring that the API of each microservice makes sense on its own is important if we wish to avoid blurring the lines between services. Not doing so may mean that independent work and deployments aren't allowed.

The dependencies between the requests and microservices should also be considered so that the work can be arranged back to front. This means preparing the new feature that will add extra data or capabilities but keeping the old behavior by default. After doing this, a new feature that uses this extra data can be deployed. This way of working ensures backward compatibility at any given time.

We will look at the features that affect multiple microservices in more detail in `Chapter 11`, *Handling Change, Dependencies, and Secrets in the System*. We'll also learn how to coordinate work and dependencies in more detail.

Going back to our previous example, to add a user's mention to their thoughts, we need to make the Thoughts Backend capable of dealing with optional references to users. This is a self-contained task that won't affect the existing functionality. It can be deployed and tested.

Then, we can make the corresponding changes in the Frontend to allow external users to interact with it through an HTML interface.

As we discussed in `Chapter 1`, *Making the Move – Design, Plan, and Execute*, it is crucial for any microservice architecture that we can deploy services independently. This allows us to test services independently and avoid any overhead that requires complicated deployments that make it difficult for us to debug and roll back if errors occur.

If different teams work independently on different microservices, then they will also require coordination.

In the next section, we will learn how to implement a feature in a single microservice.

Implementing a feature

Once we have the definition of an independent technical feature, it can be implemented.

Defining a technical feature in a clear manner can be challenging. Remember that a single feature may need to be further subdivided into smaller tasks. However, as we mentioned previously, the objective here is not to structure our tasks.

Start your task by creating a new Git branch. The code can be changed to reflect the new feature in this branch. As we saw in `Chapter 2`, *Creating a REST Service with Python*, and `Chapter 3`, *Build, Run, and Test Your Service Using Docker*, unit tests can be run to ensure that this work isn't breaking the build.

As we described in `Chapter 3`, *Build, Run, and Test Your Service Using Docker*, in the *Operating with an immutable container* section, we can use `pytest` arguments to run subsets of tests to speed up development, thereby enabling quick feedback when running tests. Make sure you use it.

How this functionality works in relation to the whole system can be checked through the deployment of the local cluster. This starts other microservices that may be affected by the work in this branch, but it helps ensure that the current work is not breaking any existing calls that affect other microservices.

Based on the pipeline, any commit that's pushed to Git will run all its tests. This will detect problems early and ensure that the build is correct before it's merged with the main branch.

While this is in progress, we can use a pull request to review the changes between the main branch and the new features. We can check our GitHub configuration to ensure that the code is in good shape before we merge it.

Once the feature is ready and has been merged with the main branch, a new tag should be created to allow for its deployment. As part of the configured pipeline, this tag will trigger a build that produces an image in the registry and label the image with the same tag. The tag and image are immutable, so we can be sure that the code won't change between the different environments. You can roll forward and back with confidence that the code will be the exact same code that was defined in the tag.

As we saw in `Chapter 8`, *Using GitOps Principles*, the tag can be deployed by following GitOps principles. The deployment is described in Kubernetes configuration files, under Git control, and is reviewed in a pull request that needs to be approved. Once the pull request has been merged with the main branch, it will be deployed by Flux automatically, as we described in `Chapter 8`, *Using GitOps Principles*, in the *Setting up Flux to control the Kubernetes cluster* section. At that time, the feature is available in the cluster.

Let's recap on this life cycle, from the description of the technical request to when it's deployed into a cluster:

This is a more complete version of the Flow that we introduced in `Chapter 4`, *Creating a Pipeline and Workflow*.

1. The technical request is ready to be implemented into a single microservice.
2. A new feature branch is created.
3. The microservice's code is changed in this branch until the feature is ready.
4. A pull request, which is used to merge the feature branch into the main branch, is created. This, as described in `Chapter 4`, *Creating a Pipeline and Workflow*, in the *Understanding the continuous integration practices* section, runs the CI process to ensure that it is of a high quality.
5. The pull request is reviewed, approved, and merged into the main branch.
6. A new tag is created.
7. A deployment branch is created in the GitOps repository that changes the version of the microservice to the new tag.
8. A pull request, which is used to merge this deployment branch, is created. Then, it's reviewed and merged.
9. Once the code has been merged, the cluster automatically releases the new version of the microservice.
10. Finally, the new feature is available in the cluster!

This is a simplified version of the life cycle; in reality, it may be more complicated. Later in this chapter, we will look at a situation where the life cycle needs to be deployed to more than one cluster.

In the next section, we will look at some recommendations when it comes to reviewing and approving pull requests.

Reviewing and approving a new feature

As specified by the pipeline model we described in `Chapter 4`, *Creating a Pipeline and Workflow*, the candidate code moves through a series of stages, stopping if there's a problem.

As we mentioned previously, reviewing using GitHub pull requests works if we wish to introduce new features to the code of our microservices, as well as if we wish to deploy those changes into clusters through GitOps practices.

In both cases, we can check this automatically through automated tests and processes. However, there's a final step that requires manual intervention: knowledge transfer and an extra pair of eyes. Once the reviewer thinks the new feature is ready, they can approve it.

The tools are the same, although the reviewing process works a little differently. This is because the objectives aren't the same. For feature code, the review is more open to discussion until it's approved and merged into the main branch. On the other hand, reviewing and approving a release is typically more straightforward and faster.

Let's start by learning how to review feature code.

Reviewing feature code

Code reviews can be initiated while the feature is being developed and a request has been opened to merge it. As we've already seen, in GitHub, code can be reviewed while it's in the **pull request** stage.

A code review is basically a shaped discussion about the code and the new feature; that is, we're checking the code before it is introduced to the main branch. This presents us with the opportunity to improve the feature while it is in development, and before it becomes a component of the system.

Here, a member of the team may read the code that hasn't been merged and give the author some feedback. This can go back and forth until the reviewer thinks that the code is ready to be merged and approves it. In essence, someone else, other than the author of the feature, needs to agree that the new code meets the required standards.

Code bases grow over time and their components can help each other out. Merging code into the main branch states that you fully accept that the new code will be maintained by the team as part of the code base.

The code may need to be approved by one or more people, or by specific people.

In GitHub, you can enable code owners. These are engineers who are responsible for approving repositories, or parts of repositories. Check out the GitHub documentation for more information: `https://help.github.com/en/articles/about-code-owners`.

Code reviewing is a quite common process these days, and the popularity and ease of using pull requests in GitHub has spread. Most developers are familiar with this process.

Implementing a good feedback culture is more difficult than it looks, though. Writing code is a deeply personal experience; no two people will write the same code. For developers, having your code criticized by others can be a difficult experience unless there are clear rules.

Here are some suggestions:

- Tell your reviewers what they should look for. Make a point of following a checklist. This helps develop a culture within the team so that they care about shared core values. This also helps junior developers know what to look for. This may change from team to team, but here are some examples:
    - There are new tests.
    - Error conditions are tested.
    - The documentation is properly updated.
    - Any new endpoints comply with standards.
    - Architectural diagrams are updated.
- Reviewing code is not the same thing as writing the code. There will always be discrepancies (for example, this variable name could be changed), but what needs to be reviewed is whether such changes need to be implemented. Nitpicking will erode the trust between team members.

- The bigger the code to review, the more difficult it is to do. It's better to work in small increments, which works well with the principles of continuous integration.
- All of the code should be reviewed on equal grounds. This includes the code of senior developers, and junior developers should be encouraged to leave honest feedback. This helps ownership of the code and its fairness to grow.
- A code review is a conversation. A comment doesn't necessarily mean that the reviewer's feedback has to be implemented without you questioning it first. It opens a conversation about improving the code, and making clarifications and pushing back is totally fine. Sometimes, the proper way of handling a request, that is, to change a part of the code, is to leave a comment explaining why this was done in a particular way.
- Reviews help spread knowledge about the code base. This isn't a silver bullet, though. Code reviews tend to fall into tunnel vision, where only small issues such as typos, and local snippets of code, are looked at, and not the bigger elements. This is why it's important to implement features in small increments: to help those around you digest change.
- It's important to leave appreciative comments. Create a culture that appreciates well-written code. Only highlighting what's bad makes the review process miserable for the author.
- Criticism should be addressed to the code, not to the coder. Ensure that your review is civil. In this step, we want to ensure that the code is of a high quality; you don't want to make yourself, as the reviewer, look superior.

Code reviews can be stressful for those who aren't used to them. Some companies are creating principles and ideas to make this process less painful. A good example can be found at `https://www.recurse.com/social-rules`. Don't be afraid to define and share your own principles.

- It's important that code can be approved at all times, even when someone in the team is on holiday or sick. Ensure that you grant approval to multiple members of the team so that the approval process itself isn't a bottleneck.

When you start producing code reviews, ensure that the team leaders keep these considerations in mind and emphasize why all the code is reviewed.

It is worth emphasizing how code reviews are not a technological solution, but a people-related one. As such, they can suffer from people-related problems such as big egos, adversarial discussions, or non-productive debates.

The microservice architecture is fit for big systems that have multiple people working on them. Teamwork is crucial. Part of that is ensuring that the code doesn't belong to a single person, but to a whole team. Code reviews are a great tool to that end, but be sure to actively look for healthy ones.

Over time, a consensus will develop and a lot of code will be developed consistently. In a healthy team, the amount of time that's spent on reviews should reduce.

Over time, the team will perform code reviews routinely, but setting these foundations may be complicated in the beginning. Ensure that you allow time to introduce them. As we mentioned previously, once the feature is ready, we need to go ahead and approve it. Approving code for a new feature and merging it into the main branch is the final stage of the feature review, but it still needs to be released. Releases are under code control, and they need to be reviewed as well.

Approving releases

Using GitOps principles allows us to enable the same review and approval methods so that we can make changes in the Kubernetes infrastructure. As we mentioned previously, the fact that the infrastructure is defined by YAML files in Kubernetes allows us to control these changes.

Any change that's made over a Kubernetes cluster can be subjected to a pull request and review method. This makes approving a release to a cluster an easy process.

This helps in minimizing problems, since more members of the team are involved in the changes and the knowledge that they have of the infrastructure is better. This works well with the DevOps principles of allowing teams to take control over their own deployment and infrastructure.

However, infrastructure changes in GitOps tend to be easier to review than regular code reviews. In general terms, they are done in very small increments, and most changes will be so straightforward that the probability of generating debate is minimal.

As a general rule, try to make any infrastructure change as small as possible. Infrastructure changes are riskier as errors may bring down important parts of it. The smaller the change, the smaller the risk and the easier it is to diagnose any issues.

All the suggestions we made about code reviews have a role to play, too. The most important one is to include some guidelines that reference critical parts of the infrastructure.

Some sections of the infrastructure may be under the GitHub code owner's protection. This makes it mandatory for certain engineers to approve changes to critical parts of the infrastructure. Take a look at the documentation for more information: `https://help.github.com/en/articles/about-code-owners`.

Since the infrastructure is defined as code stored in GitHub, this also makes it easy to copy the infrastructure, thereby greatly simplifying the generation of multiple environments.

Setting up multiple environments

The ease of creating, copying, and removing namespaces under Kubernetes greatly reduces the previous burden of keeping multiple copies of environments to replicate the underlying infrastructure. You can use this to your advantage.

Based on the GitOps principles we mentioned earlier, we can define new namespaces to generate a new cluster. We can either use another branch (for example, use the `master` branch for the production cluster and `demo` for the demo cluster) or copy the files containing the cluster definition and change the namespaces.

You can use different physical Kubernetes clusters for different purposes. It's better to leave the production cluster as not being shared with any other environment to reduce risks. However, every other environment could live in the same cluster, which won't affect external customers.

Some feature requests are evidence enough that the development team will know exactly what to do, such as with bug reports. However, others may require a bit more of testing and communication while they're in development to ensure that they fulfill a requirement. This may be the case when we're checking that a new feature is actually useful to the intended external user, or could be a more explorative feature. In this case, we need to call an external party, that is, the ultimate approver of the feature: the *stakeholder*.

A stakeholder is a term from project management that specifies a third party, that is, the final user of a product or a user who's impacted by it. Here, we use the term to specify someone who's interested in a feature but external to the team, so they can't define the feature requirements from within. A stakeholder can be, for example, a manager, a customer, the CEO of the company, or a user of an internal tool.

Any developer who has had to deal with a fuzzily defined request from a stakeholder, such as *allow search by name*, has had to tweak it: *no, not by the first name, by surname*.

Ensure that you define a proper end to these kinds of tasks. Stakeholder feedback can be endless if it's allowed to run without limits. Define what is and is not included in it, as well as any deadlines, beforehand.

To run tests and to ensure that the feature that's under development is going in the right direction, you can create one or more demo environments where you will deploy work in progress before it is merged into the main branch. This will help us share this work with stakeholders so that they can give us feedback before the feature is completed, without us having to release it in a production environment.

As we saw in the previous chapters, generating a new environment in Kubernetes is easy. We need to create a new namespace and then replicate the production definition of the cluster, thereby changing the namespace. This will create a copy of the environment.

Changing the specific version of the microservice under development will allow us to create a working version of it. Newer versions can be deployed as usual in this demo environment.

This is a simplified version. You may need to make changes between the production environment and demo environments, such as the number of replicas, and database setup. In such cases, a *template environment* could be used as a reference so that it's ready to be copied.

Other environments, such as staging, can be created in a similar fashion, with the aim of creating tests that ensure that the code that's been deployed into production will work as expected. These tests can be automatic, but they can also be manual if we want to check that the user experience is adequate.

A staging environment is a setup that works as a replica that's as faithful as possible to the production environment, which means we can run tests to provide assurance that the deployment in production will work. Staging normally helps us verify that the deployment process will be correct, as well as any final tests.

Staging environments are typically very expensive to run. After all, they are a copy of a production environment. With Kubernetes, you can replicate the production environment easily and reduce the required physical infrastructure. You can even start and stop it when it is not in use to reduce costs.

You can use multiple environments to create a cascading structure of deployment in a similar fashion. This means that a tag needs to be deployed into the staging environment and be approved before it goes into the production environment.

Let's now look at how we can deal with this structure from the point of view of the developers.

Scaling the workflow and making it work

Some of the challenges of implementing this way of working include creating a culture that provides an adequate feedback loop and checking new code carefully while reviewing it quickly. Waiting for a review is a blocked state that stops the developer from implementing the feature that's being reviewed.

While this waiting time can be used for other purposes, not being able to progress will quickly lower productivity. Either the developer will keep a few features in parallel, which is highly problematic from the context switch perspective, or they need to wait and roll their thumbs until the review is completed.

The context switch is probably the most serious killer of productivity. One of the keys to keeping your team's productivity high is being able to start and finish a task. If the task is small enough, it will be finished quickly, so swapping between projects is easier. However, working on two or more tasks at the same time is a very bad practice.

If this happens often, try to divide your tasks into smaller chunks.

To be able to balance thoroughly reviewing the code and reducing the blocking time, there are some elements to keep in mind.

Reviewing and approving is done by the whole team

There need to be sufficient reviewers available at all times. If just the developers are experienced, reviews may end up only being done by the single most senior person on the team, for example, the team lead. Though this person may be the better reviewer in principle, in the long run, this structure will compromise the team as the reviewer won't be able to do anything else. Making progress in the development and release stages will also be blocked if the reviewer is unavailable for any reason, such as due to illness or if they're on holiday.

Instead, make the whole team capable of reviewing their peer's code. Even though a senior contributor takes a more proactive role in teaching the rest of the team on how to review, after a while, most reviews shouldn't require their assistance.

Initially, though implementing this process requires active mentoring, this will typically be lead by the senior members of the team. Reviewing code is a trainable ability with the objective that, after a period of time, everyone is capable of running a review and allowed to approve pull requests.

The same process is followed for deployment pull requests. Eventually, everyone in the team, or at least a significant number of members, should be able to deploy a release. The initial main reviewer could be someone different, though.

It is likely that the best candidate to review a release is an expert on how the Kubernetes infrastructure is configured, but not an expert on microservice code.

Understanding that not every approval is the same

Remember that the different stages of a feature are not equally critical. The code review's early process is about ensuring that the code is readable and that it keeps the quality standards. In the early stages, the code will have a relatively high number of comments and there will be more to discuss since more elements will need to be tweaked.

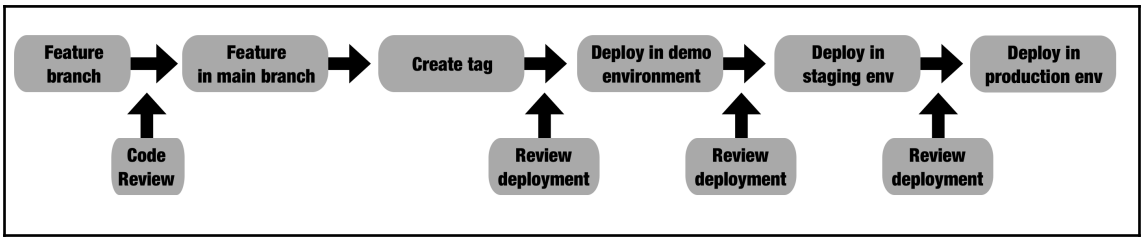
A big part of reviews is creating code that is *understandable enough* that other members of the team understand it. Although some people claim that code reviews make everyone aware of the changes that other members of the team are implementing, in my experience, reviewers are not that aware of specific features.

A good review, however, will ensure that nothing cryptic is being introduced into the code base and that the core elements are respected (elements such as introducing tests, keeping documentation up to date, and keeping code readable). As we suggested previously in this chapter, try to create an explicit list of things to check. It will help you make the reviews and code more consistent.

The deployment stages of a new feature only require that we check that the version of the microservice changes and that the remainder of the infrastructure is intact. These will typically be very small; most of them double-check that there are no typos and that the microservice to be changed is the correct one.

Defining a clear path for releases

Having a simple and clear process helps everyone involved have a clear understanding of how a feature moves from being developed to being released into the production environment. For example, based on the ideas we've discussed, we could end up with a deployment path that's similar to the one shown in the following diagram:



For each of these steps, we need to validate that the step is correct. As we saw in `Chapter 4`, *Creating a Pipeline and Workflow*, automatic tests ensure that anything that's merged into the main branch doesn't break the existing builds. This covers the preceding diagram up to the **Create tag** step.

Equally, there could be a way of validating that a deployment has been successful after it's been applied. The following are a few ideas regarding this:

- Manual tests, to check that the deployed microservice works as expected
- Automated tests, such as the ones described in `Chapter 4`, *Creating a Pipeline and Workflow*
- Check that the image to be deployed has been correctly deployed using Kubernetes tools or a version API

Once one deployment stage has been successfully completed, the next can be started.

Performing a deployment in non-production environments minimizes the risk of breaking a production as it will ensure that the deployment process is correct. The process needs to be fast enough to allow quick deployments, thus making them as small as possible.

The full process from merging into the main branch until the new version is released into the production environment should take less than a few hours, but ideally less than that.

If more time is required, the process is probably too heavy.

Small, frequent deployments will minimize the risk of breaking the production environment. In some exceptional cases, the regular process may be slow, and an emergency procedure should be used.

Emergency releases

Let's imagine that there's a critical bug in production and that it needs to be addressed as fast as possible. For these exceptional cases, it is fine to define an emergency process beforehand.

This emergency process may involve speeding up reviews or even skipping them completely. This may include skipping intermediate releases (such as not deploying to a demo environment before hand). Ensure that you explicitly define when this process is required and ensure it's used in emergency situations only.

If your regular deployment process is fast enough, then there's no need for an emergency process. This is an excellent reason to try to increase deployment times.

A rollback is a good example of such a situation. To revert the deployment of a microservice that had a critical bug introduced to it in the last version, rolling back and returning to the previous version only in production, without affecting anything else, is a reasonable process.

Note how, here, we're reducing the risk of making a quick change with the assurance that the version that's been rolled back has already been deployed before. This is an excellent example of how an emergency procedure may work and reduce risks.

Use your common sense when it comes to detecting exceptional situations and discuss how to deal with them with your team beforehand. We will talk about retrospectives in `Chapter 12`, *Collaborating and Communicating across Teams.*

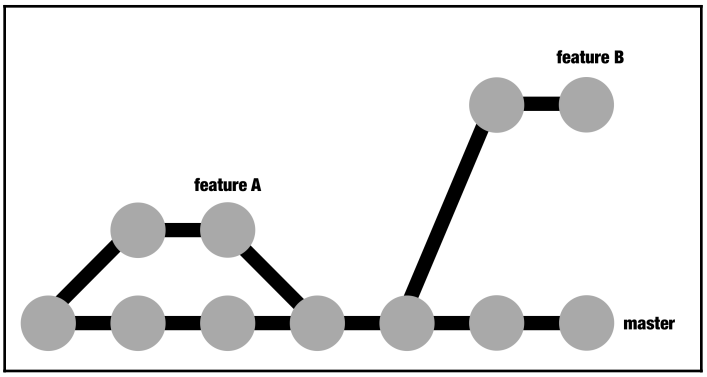
Releasing frequently and adding feature flags

While rollbacks are possible, as we've just seen, the general consensus should be that each new deployment moves forward. The code for a new release contains the code from the previous release, plus some small changes. Following the Git way of operating, we work on a single branch (the main branch) that gets pushed forward.

This means having to avoid several active long-lived branches. This model is called *trunk-based development* and it's the recommended way of working to aim for continuous integration. In a trunk-based development, feature branches are short-lived and are always merged with the main branch (or trunk), which is normally called the `master` in Git.

Trunk-based development avoids issues when we have long-lived branches that diverge from the main one, thus making the integration of several components complicated. The basis for continuous integration is to be able to always have code that can be released in small increments. This model takes "trunk" as the reference for the releases.

In the following diagram, we can see how **feature A** was merged to the **master** branch and how **feature B** is still in progress. Any releases will come from the **master** branch:



If **feature A** were to introduce a bug, a new bugfix branch will branch off from **master** and will be merged back. Note how the structure is to keep going forward.

For this system to work, feature branches need to be short-lived – typically for only a few days. This makes merges easy and allows for small incremental changes, which are the key to continuous integration.

Using feature flags

Sometimes, there are some features that, by design, require big/drastic changes to be made in one go, such as a new UI interface. The kind of short, quick iteration cycle of slowly adding small features that continuous integration advocates doesn't work in these frequent release situations. The new interface needs to include all the elements in one go, or it will look weird.

You can use feature flags when you want to keep working in a small incremental way and, at the same time, delay the activation of a feature until it's ready.

Feature flags are configuration elements that enable or disable a particular feature. This allows you to change the behavior of the microservice with a configuration change, which acts as a switch.

In Kubernetes, we use the `deployment.yaml` file to describe the environment variables, as well as ConfigMaps. We will discuss ConfigMaps in `Chapter 11`, *Handling Change, Dependencies, and Secrets in the System*.

The configuration is tied to each individual environment. This makes it possible for us to present a feature in a particular environment and not in another, while the code base remains the same.

For example, a new interface can be slowly developed and protected under a feature flag. Some environments, such as demo environments, can still be active so that internal feedback can be gathered, but this won't be displayed in the production environment.

Once the new interface is ready, small changes can be made; for example, we can change the configuration parameter to enable it. This may look like a big change externally, but it can easily be reverted if we swap back to the parameter.

Feature flags are useful when we're dealing with externally accessible services. Internal services can add more features without any issue since they'll only be called by other microservices in the system.

Internal microservices are normally okay with adding new features. Here, backward compatibility is respected. Externally accessible features sometimes require us to replace a feature with another for reasons including interface changes or the deprecation of products.

A related approach is to roll a feature to a subset of users. This can be a predefined set of users, such as users who have enrolled in a beta program to receive early access to features or a random sample so that they can detect problems early, ahead of a global release.

Some big companies use regional access as well, where some features are enabled in certain countries first.

Once the feature flag has been activated, any deprecated features can be removed and cleaned up, so there's no old code that's not going to be used.

Dealing with database migrations

Database migrations are changes that are made to the persistent data that's stored in a particular environment (normally, in one or more databases). Most of the time, this means changing the database schema, but there are others.

The data in a production environment is the most important asset in a running system. Extra care is advised for database migrations.

In certain cases, a migration may lock a table for a certain period of time, thereby rendering the system unusable. Ensure that you test your migrations properly in order to avoid or at least prepare for these cases.

Though database migrations may technically be reversible, doing so is very expensive in terms of development time. For example, adding and removing a column could be simple, but once the column is in operation, the column will contain data that shouldn't be removed.

To be able to work seamlessly in the event of data migration, you need to detach it from the code that is going to call it and follow these steps:

1. Design a database migration in a way that doesn't interfere with the current code. For example, adding a table or column to the database is safe since the old code will ignore it.
2. Perform database migration. This makes the required changes while the existing code keeps operating without interruption.
3. Now, the code can be deployed. Once it has been deployed, it will start using the advantages of the new database definition. If there's a problem, the code can be rolled back to a previous version.

This implies that we need to create two deployments:

- One for the migration
- Another for the code that uses this migration

Migration deployment may be similar to code deployment. Maybe there's a microservice running the migrations, or maybe it's a script doing all the work. Most frameworks will have a way of making migrations to ensure that a migration isn't applied twice.

For example, for SQLAlchemy, there's a tool called Alembic (`https://alembic.sqlalchemy.org/en/latest/`) that we can use to generate and run migrations.

However, there is an alternative operation: try to apply the migrations to the microservice that will make use of them. When dealing with a production environment, this is a bad idea as this will slow start up times in all situations, regardless of whether a migration is occurring. Also, it won't check that the code can be safely rolled back and works with the previous version of the database.

Working with two independent deployments is obviously a bit more restrictive than changing the database freely, but it ensures that each step forward is solid and that the service is uninterrupted. It's more deliberate. For example, to rename a column, we would follow these steps:

1. First, we would deploy a migration that creates a new column with the new column name, thereby copying the data from the old column. The code reads and writes from the old column.
2. Then, we would deploy the new code that reads from the old column and writes to both. During the release process, any writes from the old code to the old column will be read correctly.
3. After, we would create another migration that copies the data from the old one to the new one. This ensures that any transient copy is correctly applied. At this point, any new data still goes to both columns.
4. Then, we would deploy code that reads and writes to the new column, ignoring the old one.
5. Finally, we would implement a migration to drop the old column. At this point, the old column doesn't contain relevant data and can be safely deleted. It won't affect the code.

This is a deliberate example of a long process, but in most cases, such a long process won't be required. However, at no point in any of these steps is there any inconsistency. If there's a problem in one of the stages, we can revert to the previous stage – it will still work until a fix is put in place.

The main objective is to avoid having transient states where the database won't work with the currently deployed code.

Summary

In this chapter, we talked about the flow of a team, from starting a new feature to deploying it into a production environment.

We started by talking about the key points of feature requests when we're working in a microservices architecture. We introduced requests that affect multiple microservices and learned how to structure the work so that the service isn't interrupted.

We talked about the elements that make a good review and approval process, as well as how GitHub pull requests help us do this. Using GitOps practices to control the infrastructure makes deployments to be reviewed straightforward.

Then, we discussed how working with Kubernetes and GitOps helps us create multiple environments and how we can use them to our advantage when dealing with demo and staging environments to test deployments and to present features in a controlled environment before they go to production.

After this, we talked about how we can make a team have a global view of the entire life cycle, from feature request to deployment and being able to follow the full path quickly. We learned how to clarify these steps and how to make the team responsible for reviewing and approving its own code, which allows developers to take full ownership of the development cycle.

We also talked about the issues that may occur when we're dealing with database migrations and explained how to proceed with this special kind of deployment, which isn't easy to roll back.

In the next chapter, we will talk about live systems and how to enable elements such as metrics and logs so that we can detect problems and bugs that occur in a production environment and have enough information to remediate them as quickly and proactively as possible.

Questions

1. As a new business feature is received, what analysis do we need to perform in a system working under a microservice architecture?
2. If a feature requires two or more microservices to be changed, how do we decide which one should be changed first?
3. How does Kubernetes help us set up multiple environments?
4. How does a code review work?
5. What is the main bottleneck for code reviews?
6. Under GitOps principles, are the reviews for deployment different from code reviews?
7. Why is it important to have a clear path to deployment once a feature is ready to be merged into the main branch?
8. Why are database migrations different from regular code deployments?

Further reading

To learn more about agile practices and introducing them to a team, take a look at the following books:

- *The Agile Developer's Handbook* (`https://www.packtpub.com/eu/web-development/agile-developers-handbook`)
- *Agile Technical Practices Distilled* (`https://www.packtpub.com/eu/business-other/agile-technical-practices-distilled`)

If you're using JIRA in your organization, reading *Hands-On Agile Software Development with JIRA* (`https://www.packtpub.com/eu/application-development/hands-agile-software-development-jira`) can help you get better use out of the tool when you're working with agile practices.

4
Section 4: Production-Ready System – Making It Work in Real-Life Environments

The final section of the book focuses on some of the elements that make a live system work long term in real life, from the observability of the system, which is critical for detecting and fixing problems quickly, to handling the configuration that affects the whole system and includes techniques for ensuring that the different teams collaborate and develop systems in a coordinated fashion.

The first chapter of this section deals with how to discover an operation on a live cluster in order to detect usage and associated problems. This chapter introduces the concept of observability and the two main tools for supporting it: logs and metrics. It covers how to include them properly in a Kubernetes cluster.

The second chapter of this section deals with a configuration that is shared across different microservices and how to work with dependencies between services. It also shows how to deal with secrets in real life: configuration parameters that contain sensible information, such as security keys and certificates.

The third chapter of this section describes common problems on inter-team communication when working in microservice architectures and how to deal with them, including how to create a shared vision across the whole organization, how the division in teams affects the different APIs, and how to release new features across teams.

This section comprises the following chapters:

- `Chapter 10`, *Monitoring Logs and Metrics*
- `Chapter 11`, *Handling Change, Dependencies, and Secrets in the System*
- `Chapter 12`, *Collaboration and Communication across Teams*

10
Monitoring Logs and Metrics

In real-life operations, the ability to quickly detect and debug a problem is critical. In this chapter, we will discuss the two most important tools we can use to discover what's happening in a production cluster processing a high number of requests. The first tool is logs, which help us to understand what's happening within a single request, while the other tool is metrics, which categorizes the aggregated performance of the system.

The following topics will be covered in this chapter:

- Observability of a live system
- Setting up logs
- Detecting problems through logs
- Setting up metrics
- Being proactive

By the end of this chapter, you'll know how to add logs so that they are available to detect problems and how to add and plot metrics and understand the differences between both of them.

Technical requirements

We will be using the example system for this chapter and tweaking it to include centralized logging and metrics. The code for this chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter10`.

To install the cluster, you need to build each individual microservice:

```
$ cd Chapter10/microservices/
$ cd frontend
$ docker-compose build
...
$ cd thoughts_backend
$ docker-compose build
...
$ cd users_backend
$ docker-compose build
...
```

The microservices in this chapter are the same ones that we introduced previously, but they add extra log and metrics configuration.

Now, we need to create the example namespace and start the Kubernetes cluster using the `find` configuration in the `Chapter10/kubernetes` subdirectory:

```
$ cd Chapter10/kubernetes
$ kubectl create namespace example
$ kubectl apply --recursive -f .
...
```

To be able to access the different services, you need to update your `/etc/hosts` file so that it includes the following lines of code:

```
127.0.0.1 thoughts.example.local
127.0.0.1 users.example.local
127.0.0.1 frontend.example.local
127.0.0.1 syslog.example.local
127.0.0.1 prometheus.example.local
127.0.0.1 grafana.example.local
```

With that, you will be able to access the logs and metrics for this chapter.

Observability of a live system

Observability is the capability of knowing what's going on in a live system. We can deal with low-observability systems, where we have no way of knowing what's going on, or high-observability systems, where we can infer the events and internal state from the outside through tools.

Observability is a property of the system itself. Typically, monitoring is the action of obtaining information about the current or past state of the system. It's all a bit of a naming debate, but you monitor the system to collect the observable parts of it.

For the most part, monitoring is easy. There are great tools out there that can help us capture and analyze information and present it in all kinds of ways. However, the system needs to expose the relevant information so that it can be collected.

Exposing the correct amount of information is difficult. Too much information will produce a lot of noise that will hide the relevant signal. Too little information will not be enough to detect problems. In this chapter, we will look at different strategies to combat this, but every system will have to explore and discover this on its own. Expect to experiment and make changes in your own system!

Distributed systems, such as the ones that follow a microservice architecture, also present problems as the complexity of the system can make it difficult to understand its internal state. Behavior can be also unpredictable in some circumstances. This kind of system at scale is inherently never completely healthy; there will always be minor problems here and there. You need to develop a priority system to determine what problems require immediate action and which ones can be solved at a later stage.

The main tools for the observability of microservices are **logs** and **metrics**. They are well-understood and used by the community, and there are plenty of tools that greatly simplify their usage, both as packages that can be installed locally and as cloud services that can help with data retention and the reduction of maintenance costs.

Using cloud services for monitoring will save you from maintenance costs. We will talk about this later in the *Setting up logs* and *Setting up metrics* sections.

Another alternative when it comes to observability is services such as Data Dog (`https://www.datadoghq.com/`) and New Relic (`https://newrelic.com/`). They receive events – normally logs – and are able to derive metrics from there.

The most important details of the state of the cluster can be checked through `kubectl`, as we saw in previous chapters. This will include details such as the versions that have been deployed, restarts, pulling images, and so on.

For production environments, it may be good to deploy a web-based tool to display this kind of information. Check out Weave Scope, an open source tool that shows data in a web page similar to the one that can be obtained with `kubectl`, but in a nicer and more graphical way. You can find out more about this tool here: `https://www.weave.works/oss/scope/`.

Logs and metrics have different objectives, and both can be intricate. We will look at some common usages of them in this book.

Understanding logs

Logs track unique events that occur in the system. Each log stores a message, which is produced when a specific part of the code is executed. Logs can be totally generic (*function X is called*) or include specific details (*function X is called with parameter A*).

The most common format for logs is to generate them as plain strings. This is very flexible, and normally log-related tools work with text searches.

Each log includes some metadata about who produced the log, what time it was created, and more. This is also normally encoded as text, at the beginning of the log. A standard format helps with sorting and filtering.

Logs also include a severity level. This allows for categorization so that we can capture the importance of the messages. The severity level can be, in order of importance, `DEBUG`, `INFO`, `WARNING`, or `ERROR`. This severity allows us to filter out unimportant logs and determine actions that we should take. The logging facility can be configured to set a threshold; less severe logs are ignored.

There are many severity levels, and you can define custom intermediate levels if you wish. However, this isn't very useful except in very specific situations. Later in this chapter, in the *Detecting problems through logs* section, we will describe how to set a strategy per level; too many levels can add confusion.

In a web service environment, most of the logs will be generated as part of the response for a web request. This means that a request will arrive at the system, be processed, and return a value. Several logs will be generated along the way. Keep in mind that, in a system under load, multiple requests will be happening simultaneously, so the logs from multiple requests will also be generated simultaneously. For example, note how the second log comes from a different IP:

```
Aug 15 00:15:15.100 10.1.0.90 INFO app: REQUEST GET /endpoint
Aug 15 00:15:15.153 10.1.0.92 INFO api: REQUEST GET /api/endpoint
Aug 15 00:15:15.175 10.1.0.90 INFO app: RESPONSE TIME 4 ms
Aug 15 00:15:15.210 10.1.0.90 INFO app: RESPONSE STATUS 200
```

A common request ID can be added to group all the related logs that have been produced for a single request. We will see how to do this later in this chapter.

Each individual log can be relatively big and, in aggregate, use significant disk space. Logs can quickly grow out of proportion in a system under load. The different log systems allow us to tweak their retention time, which means that we only keep them for a certain amount of time. Finding the balance between keeping logs to see what happened in the past and using a sane amount of space is important.

Be sure to check the retention policies when enabling any new log service, whether it be local or cloud-based. You won't be able to analyze what happened before the time window. Double-check that the progress rate is as expected – you don't want to find out that you went unexpectedly over quota while you were tracking a bug.

Some tools allow us to use raw logs to generate aggregated results. They can count the number of times a particular log appears and generate the average times per minute or other statistics. This is expensive, though, as each log takes space. To observe this aggregated behavior, it is better to use a specific metrics system.

Understanding metrics

Metrics deal with aggregated information. They show information related not to a single event, but a group of them. This allows us to check the general status of the cluster in a better way than using logs.

We will use typical examples related to web services, mainly dealing with requests metrics, but don't feel restricted by them. You can generate your own metrics that are specific to your service!

Where a log keeps information about each individual event, metrics reduce the information to the number of times the event happens or reduce them to a value that can then be averaged or aggregated in a certain way.

This makes metrics much more lightweight than logs and allows us to plot them against time. Metrics present information such as the number of requests per minute, the average time for a request during a minute, the number of queued requests, the number of errors per minute, and so on.

The resolution of the metrics may depend on the tool that was used to aggregate them. Keep in mind that a higher resolution will require more resources. A typical resolution is 1 minute, which is small enough to present detailed information unless you have a very active system that receives 10 or more requests per second.

Capturing and analyzing information related to performance, such as the average request time, allows us to detect possible bottlenecks and act quickly in order to improve the performance of the system. This is much easier to deal with on average since a single request may not capture enough information for us to see the big picture. It also helps us predict future bottlenecks.

There are many different kinds of metrics, depending on the tool that's used. The most commonly supported ones are as follows:

- **Counter**: A trigger is generated each time something happens. This will be counted and aggregated. An example of this is the number of requests and the number of errors.
- **Gauge**: A single number that is unique. It can go up or down, but the last value overwrites the previous. An example of this is the number of requests in the queue and the number of available workers.
- **Measure**: Events that have a number associated with them. These numbers can be averaged, summed, or aggregated in some way. Compared with gauges, the difference is that previous measures are still independent; for example, when we request time in milliseconds and request size in bytes. Measures can also work as counters since their number can be important; for example, tracking the request time also counts the number of requests.

There are two main ways in which metrics work:

- Each time something happens, an event gets *pushed* toward the metrics collector.
- Each system maintains their own metrics, which are then *pulled* from the metrics system periodically.

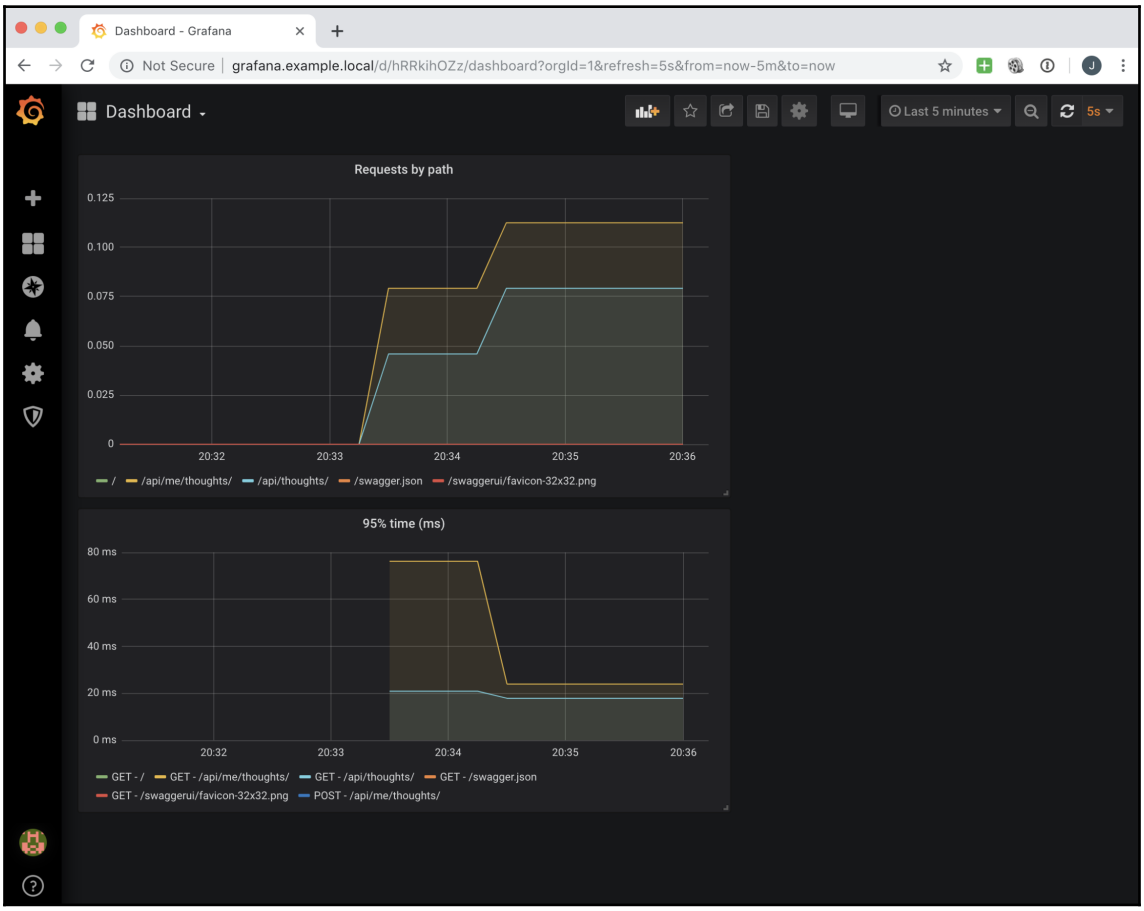
Each way has its own pros and cons. Pushing events produces higher traffic as every event needs to be sent; this can cause bottlenecks and delays. Pulling events will only sample the information and miss exactly what happened between the samples, but it's inherently more scalable.

While both approaches are used, the trend is moving toward pulling systems for metrics. They reduce the maintenance that's required for pushing systems and are much more easier to scale.

We will set up Prometheus, which uses the second approach. The most used exponent of the first approach is Graphite.

Metrics can also be combined to generate other metrics; for example, we can divide the number of requests that return errors by the total number of requests that generate error requests. Such derived metrics can help us present information in a meaningful way.

Multiple metrics can be displayed in dashboards so that we can understand the status of a service or cluster. At a glance, these graphical tools allow us to detect the general state of the system. We will set Grafana so that it displays graphical information:



Compared to logs, metrics take up much less space and they can capture a bigger window of time. It's even possible to keep metrics for the system's life. This differs compared to logs, which can never be stored for that long.

Setting up logs

We will centralize all the logs that are generated by the system into a single pod. In local development, this pod will expose all the received logs through a web interface.

The logs will be sent over the `syslog` protocol, which is the most standard way of transmitting them. There's native support for `syslog` in Python, as well as in virtually any system that deals with logging and has Unix support.

Using a single container makes it easy to aggregate logs. In production, this system should be replaced with a container that relays the received logs to a cloud service such as Loggly or Splunk.

There are multiple `syslog` servers that are capable of receiving logs and aggregating them; `syslog-ng` (`https://www.syslog-ng.com/`) and `rsyslog` (`https://www.rsyslog.com/`) are the most common ones. The simplest method is to receive the logs and to store them in a file. Let's start a container with an `rsyslog` server, which will store the received logs.

Setting up an rsyslog container

In this section, we will create our own `rsyslog` server. This is a very simple container, and you can check `docker-compose` and `Dockerfile` on GitHub for more information regarding logs (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter10/kubernetes/logs`).

We will set up logs using the UDP protocol. This is the standard protocol for `syslog`, but it's less common than the usual HTTP over TCP that's used for web development.

The main difference is that UDP is connectionless, so the log is sent and no confirmation that it has been delivered is received. This makes UDP lighter and faster, but also less reliable. If there's a problem in the network, some logs may disappear without warning.

This is normally an adequate trade-off since the number of logs is high and the implications of losing a few isn't big. `syslog` can also work over TCP, thus increasing reliability but also reducing the performance of the system.

The Dockerfile installs `rsyslog` and copies its configuration file:

```
FROM alpine:3.9

RUN apk add --update rsyslog

COPY rsyslog.conf /etc/rsyslog.d/rsyslog.conf
```

The configuration file mainly starts the server at port `5140` and stores the received files in `/var/log/syslog`:

```
# Start a UDP listen port at 5140
module(load="imudp")
input(type="imudp" port="5140")
...
# Store the received files in /var/log/syslog, and enable rotation
$outchannel log_rotation,/var/log/syslog, 5000000,/bin/rm /var/log/syslog
```

With log rotation, we set a limit on the side of the `/var/log/syslog` file so that it doesn't grow without limits.

We can build the container with the usual `docker-compose` command:

```
$ docker-compose build
Building rsyslog
...
Successfully built 560bf048c48a
Successfully tagged rsyslog:latest
```

This will create a combination of a pod, a service, and an Ingress, as we did with the other microservices, to collect logs and allow external access from a browser.

Defining the syslog pod

The `syslog` pod will contain the `rsyslog` container and another container to display the logs.

To display the logs, we will use front rail, an application that streams log files to a web server. We need to share the file across both containers in the same pod, and the simplest way to do this is through a volume.

We control the pod using a deployment. You can check the deployment configuration file at `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter10/kubernetes/logs/deployment.yaml`. Let's take a look at its most interesting parts in the following subsections.

log-volume

`log-volume` creates an empty directory that is shared across both containers:

```
    volumes:
    - emptyDir: {}
      name: log-volume
```

This allows the containers to communicate while storing information in a file. The `syslog` container will write to it while the front rail one will read from it.

syslog container

The `syslog` container starts an `rsyslogd` process:

```
  spec:
    containers:
    - name: syslog
      command:
        - rsyslogd
        - -n
        - -f
        - /etc/rsyslog.d/rsyslog.conf
      image: rsyslog:latest
      imagePullPolicy: Never
      ports:
        - containerPort: 5140
          protocol: UDP
      volumeMounts:
        - mountPath: /var/log
          name: log-volume
```

The `rsyslogd -n -f /etc/rsyslog.d/rsyslog.conf` command starts the server with the configured file we described previously. The `-n` parameter keeps the process in the foreground, thereby keeping the container running.

The UDP port `5140`, which is the defined port to receive logs, is specified, and `log-volume` is mounted to `/var/log`. Later in the file, `log-volume` will be defined.

The front rail container

The front rail container is started from the official container image:

```
    - name: frontrail
      args:
```

```
      - --ui-highlight
      - /var/log/syslog
      - -n
      - "1000"
      image: mthenw/frontail:4.6.0
      imagePullPolicy: Always
      ports:
      - containerPort: 9001
        protocol: TCP
      resources: {}
      volumeMounts:
      - mountPath: /var/log
        name: log-volume
```

We start it with the `frontrail /var/log/syslog` command, specify port `9001` (which is the one we use to access `frontrail`), and mount `/var/log`, just like we did with the `syslog` container, to share the log file.

Allowing external access

As we did with the other microservices, we will create a service and an Ingress. The service will be used by other microservices so they can send their logs. The Ingress will be used to access the web interface so that we can see the logs as they arrive.

The YAML files are on GitHub (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter10/kubernetes/logs`) in the `service.yaml` and `ingress.yaml` files, respectively.

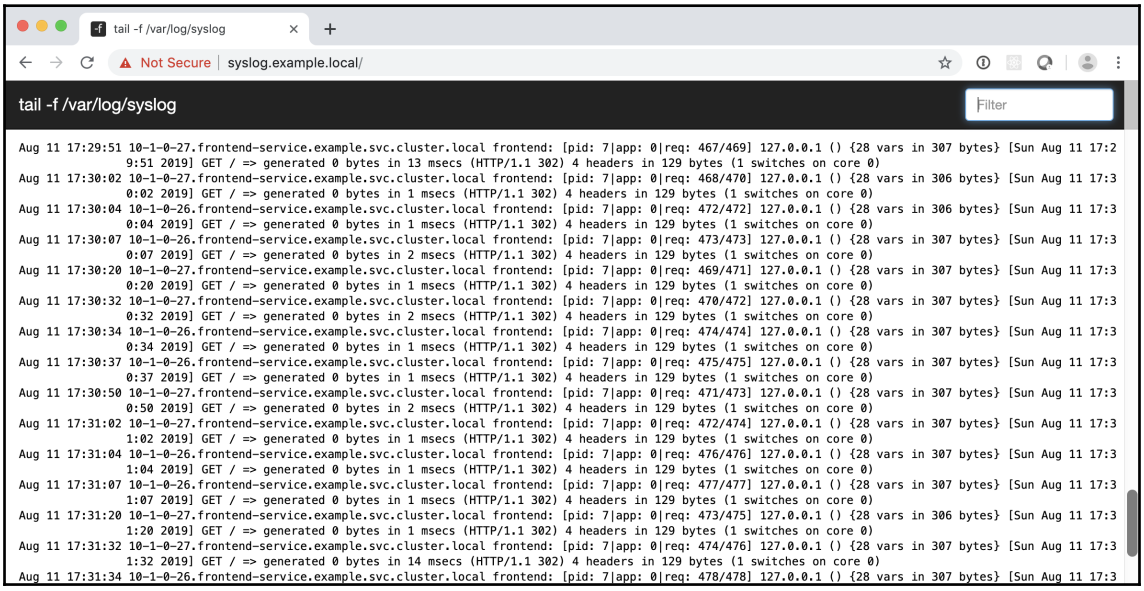
The service is very straightforward; the only peculiarity is that it has two ports – one TCP port and one UDP port – and each one connects to a different container:

```
spec:
  ports:
  - name: fronttail
    port: 9001
    protocol: TCP
    targetPort: 9001
  - name: syslog
    port: 5140
    protocol: UDP
    targetPort: 5140
```

The Ingress only exposes the front rail port, which means we can access it through the browser. Remember that the DNS needs to be added to your `/etc/host` file, as described at the start of this chapter:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: syslog-ingress
  namespace: example
spec:
  rules:
  - host: syslog.example.local
    http:
      paths:
      - backend:
          serviceName: syslog
          servicePort: 9001
        path: /
```

Going to `http://syslog.example.local` in your browser will allow you to access the front rail interface:



You can filter the logs using the box in the top-right corner.

Remember that, most of the time, logs reflect the readiness and liveness probes, as shown in the preceding screenshot. The more health checks you have in your system, the more noise you'll get.

You can filter them out at the `syslog` level by configuring the `rsyslog.conf` file, but be careful not to leave out any relevant information.

Now, we need to see how the other microservices configure and send their logs here.

Sending logs

We need to configure the microservices in uWSGI so that we can forward the logs to the logging service. We will use the Thoughts Backend as an example, even though the Frontend and Users Backend, which can be found under the `Chapter10/microservices` directory, also have this configuration enabled.

Open the `uwsgi.ini` configuration file (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter10/microservices/thoughts_backend/docker/app/uwsgi.ini`). You'll see the following line:

```
# Log to the logger container
logger = rsyslog:syslog:5140,thoughts_backend
```

This sends the logs, in `rsyslog` format, toward the `syslog` service at port `5140`. We also add the *facility*, which is where the logs come from. This adds the string to all the logs coming from this service, which helps with sorting and filtering. Each `uwsgi.ini` file should have its own facility to help with filtering.

In old systems that support the `syslog` protocol, the facility needs to fit predetermined values such as `KERN`, `LOCAL_7`, and more. But in most modern systems, this is an arbitrary string that can take any value.

Automatic logs by uWSGI are interesting, but we also need to set up our own logs for custom tracking. Let's see how.

Generating application logs

Flask automatically configures a logger for the app. We need to add a log in the following way, as shown in the `api_namespace.py` file (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter10/microservices/thoughts_backend/ThoughtsBackend/thoughts_backend/api_namespace.py#L102`):

```
from flask import current_app as app

...
if search_param:
    param = f'%{search_param}%'
    app.logger.info(f'Searching with params {param}')
    query = (query.filter(ThoughtModel.text.ilike(param)))
```

`app.logger` can call `.debug`, `.info`, `.warning`, or `.error` to generate a log. Note that `app` can be retrieved by importing `current_app`.

The logger follows the standard `logging` module in Python. It can be configured in different ways. Take a look at the `app.py` file (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter10/microservices/thoughts_backend/ThoughtsBackend/thoughts_backend/app.py`) to view the different configuration we'll be going through in the following subsections.

Dictionary configuration

The first level of logging goes through the default `dictConfig` variable. This variable is automatically defined by Flask and allows us to configure the logs in the way that's defined in the Python documentation (`https://docs.python.org/3.7/library/logging.config.html`). You can check the definition of logging in the `app.py` file:

```
from logging.config import dictConfig

dictConfig({
    'version': 1,
    'formatters': {
        'default': {
            'format': '[%(asctime)s] %(levelname)s in
                       %(module)s: %(message)s',
        }
    },
    'handlers': {
        'wsgi': {
```

```
            'class': 'logging.StreamHandler',
            'stream': 'ext://flask.logging.wsgi_errors_stream',
            'formatter': 'default'
        }
    },
    'root': {
        'level': 'INFO',
        'handlers': ['wsgi']
    }
})
```

The `dictConfig` dictionary has three main levels:

- `formatters`: This checks how the log is formatted. To define the format, you can use the automatic values that are available in the Python documentation (`https://docs.python.org/3/library/logging.html#logrecord-attributes`). This gathers information for every log.
- `handlers`: This checks where the log goes to. You can assign one or more to the loggers. We defined a handler called `wsgi` and configured it so that it goes up, toward uWSGI.
- `root`: This is the top level for logs, so anything that wasn't previously logged will refer to this level. We configure the `INFO` logging level here.

This sets up default configuration so that we don't miss any logs. However, we can create even more complex logging handlers.

Logging a request ID

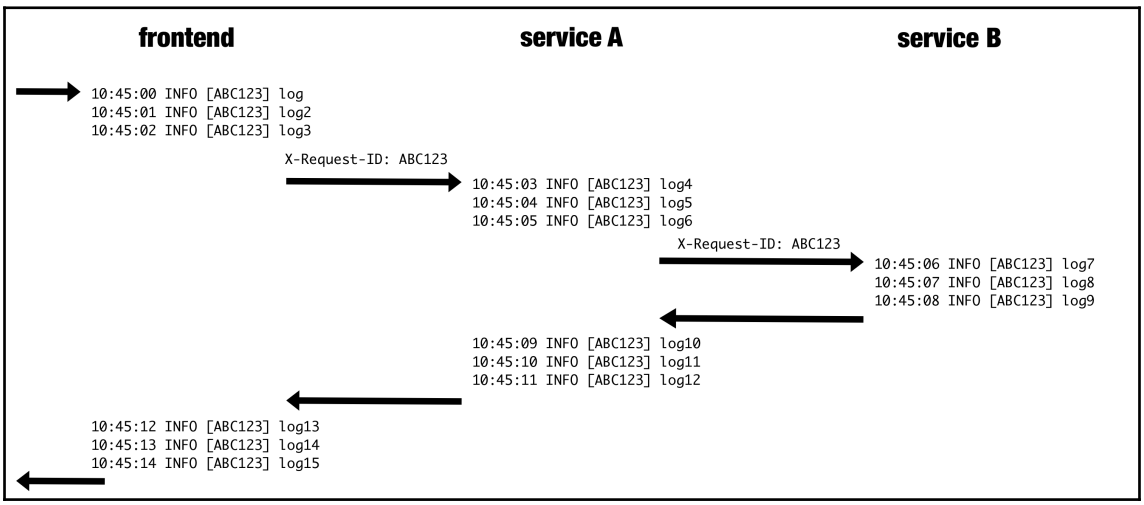
One of the problems when analyzing a large number of logs is correlating them. We need to see which ones are related to each other. One possibility is to filter logs by the pod that's generating them, which is stored at the start of the log (for example, `10-1-0-27.frontend-service.example.svc.cluster.local`). This is analogous to the host generating the logs. This process, however, is cumbersome and, in some cases, a single container can process two requests simultaneously. We need a unique identifier per request that gets added to all the logs for a single request.

To do so, we will use the `flask-request-id-header` package (`https://pypi.org/project/flask-request-id-header/`). This adds an `X-Request-ID` header (if not present) that we can use to log each individual request.

Why do we set up a header instead of storing a randomly generated value in memory for the request? This is a common pattern that allows us to inject the request ID into the backend. The request ID allows us to carry over the same request identifier through the life cycle of a request for different microservices. For example, we can generate it on the Frontend and pass it over to the Thoughts Backend so that we can trace several internal requests that have the same origin.

Although we won't be including this in our example for simplicity, as a microservices system grows, this becomes crucial for determining flows and origins. Generating a module so that we can automatically pass it over internal calls is a good investment.

The following diagram shows the flow between a **frontend** and two services. Note that the `X-Request-ID` header is not set up for the **frontend** service upon arrival and that it needs to be forwarded to any call:



We need to also send the logs straight toward the `syslog` service so that we can create a handler that does this for us.

When executing code from a script, compared to running the code in a web server, we don't use this handler. When running a script directly, we want our logs to go to the default logger we defined previously. In `create_app`, we will set up a parameter to differentiate between them.

The Python logging module has a lot of interesting features. Check out the Python documentation for more information (`https://docs.python.org/3/library/logging.html`).

Setting logs properly is trickier than it looks. Don't be discouraged and keep tweaking them until they work.

We will set up all the logging configuration in the `app.py` file. Let's break up each part of the configuration:

1. First, we will generate a formatter that appends the `request_id` so that it's available when generating logs:

```
class RequestFormatter(logging.Formatter):
    ''' Inject the HTTP_X_REQUEST_ID to format logs '''

    def format(self, record):
        record.request_id = 'NA'

        if has_request_context():
            record.request_id =
request.environ.get("HTTP_X_REQUEST_ID")

        return super().format(record)
```

   As you can see, the `HTTP_X_REQUEST_ID` header is available in the `request.environ` variable.

2. Later, in `create_app`, we will set up the handler that we append to the `application` logger:

```
# Enable RequestId
application.config['REQUEST_ID_UNIQUE_VALUE_PREFIX'] = ''
RequestID(application)

if not script:
    # For scripts, it should not connect to Syslog
    handler = logging.handlers.SysLogHandler(('syslog', 5140))
    req_format = ('[%(asctime)s] %(levelname)s [%(request_id)s] '
                  '%(module)s: %(message)s')
    handler.setFormatter(RequestFormatter(req_format))
    handler.setLevel(logging.INFO)
    application.logger.addHandler(handler)
    # Do not propagate to avoid log duplication
    application.logger.propagate = False
```

We only set up the handler if the run happens out of a script. `SysLogHandler` is included in Python. After this, we set up the format, which includes `request_id`. The formatter uses the `RequestFormatter` that we defined previously.

Here, we are hardcoding the values of the logger level to `INFO` and the `syslog` host to `syslog`, which corresponds to the service. Kubernetes will resolve this DNS correctly. Both values can be passed through environment variables, but we didn't do this here for the sake of simplicity.

The logger hasn't been propagated, so avoid sending it to the `root` logger, which will duplicate the log.

Logging each request

There are common elements in every request that we need to capture. Flask allows us to execute code before and after a request, so we can use that to log the common elements of each request. Let's learn how to do this.

From the `app.py` file, we will define the `logging_before` function:

```
from flask import current_app, g

def logging_before():
    msg = 'REQUEST {REQUEST_METHOD}
{REQUEST_URI}'.format(**request.environ)
    current_app.logger.info(msg)

    # Store the start time for the request
    g.start_time = time()
```

This creates a log with the word `REQUEST` and two essential parts of each request – the method and the URI – which come from `request.environ`. Then, they're added to an `INFO` log with the app logger.

We also use the `g` object to store the time when the request is started.

The `g` object allows us to store values through a request. We will use it to calculate the time the request is going to take.

There's the corresponding `logging_after` function as well. It gathers the time at the end of the request and calculates the difference in milliseconds:

```
def logging_after(response):
    # Get total time in milliseconds
    total_time = time() - g.start_time
    time_in_ms = int(total_time * 1000)
    msg = f'RESPONSE TIME {time_in_ms} ms'
    current_app.logger.info(msg)

    msg = f'RESPONSE STATUS {response.status_code.value}'
    current_app.logger.info(msg)

    # Store metrics
    ...

    return response
```

This will allow us to detect requests that are taking longer and will be stored in metrics, as we will see in the following section.

Then, the functions are enabled in the `create_app` function:

```
def create_app(script=False):
    ...
    application = Flask(__name__)
    application.before_request(logging_before)
    application.after_request(logging_after)
```

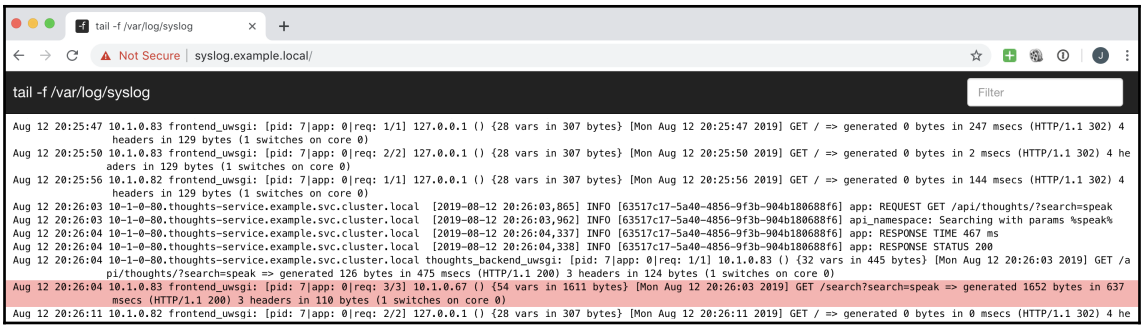
This creates a set of logs each time we generate a request.

With the logs generated, we can search for them in the `frontrail` interface.

Searching through all the logs

All the different logs from different applications will be centralized and available to search for at `http://syslog.example.local`.

If you make a call to `http://frontend.example.local/search?search=speak` to search for thoughts, you will see the corresponding Thoughts Backend in the logs, as shown in the following screenshot:



We can filter by the request ID, that is, `63517c17-5a40-4856-9f3b-904b180688f6`, to get the Thoughts Backend request logs. Just after this are the `thoughts_backend_uwsgi` and `frontend_uwsgi` request logs, which show the flow of the request.

Here, you can see all the elements we talked about previously:

- The `REQUEST` log before the request
- The `api_namespace` request, which contains app data
- The after `RESPONSE` logs, which contain the result and time

Within the code for the Thoughts Backend, we left an error on purpose. It will be triggered if a user tries to share a new thought. We will use this to learn how to debug issues through logs.

Detecting problems through logs

For any problem in your running system, there are two kinds of errors that can occur: expected errors and unexpected errors.

Detecting expected errors

Expected errors are errors that are raised by explicitly creating an `ERROR` log in the code. If an error log is being generated, this means that it reflects a situation that has been planned in advance; for example, you can't connect to the database, or there is some data stored in an old, deprecated format. We don't want this to happen, but we saw the possibility of it happening and prepared the code to deal with it. They normally describe the situation well enough that the issue is obvious, even if the solution isn't.

They are relatively easy to deal with since they describe foreseen problems.

Capturing unexpected errors

Unexpected errors are the other types of errors that can occur. Things break in unforeseen ways. Unexpected errors are normally produced by Python exceptions being raised at some point in the code and not being captured.

If logging has been properly configured, any exceptions or errors that haven't been caught will trigger an `ERROR` log, which will include the stack trace. These errors may not be immediately obvious and will require further investigation.

To help explain these errors, we introduced an exception in the code for the Thoughts Backend in the `Chapter10` code. You can check the code on GitHub (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter10/microservices/thoughts_backend/ThoughtsBackend/thoughts_backend`). This simulates an unexpected exception.

While trying to post a new thought for a logged user, we get a weird behavior and see the following error in the logs. As shown in the top-right corner of the following screenshot, we are filtering by `ERROR` to filter for problems:

As you can see, the stack trace is displayed in a single line. This may depend on how you capture and display the logs. Flask will automatically generate an HTTP response with a status code of 500. This may trigger more errors along the path if the caller isn't ready to receive a 500 response.

Then, the stack trace will let you know what broke. In this case, we can see that there's a `raise Exception` command in the `api_namespace.py` file at line **80**. This allows us to locate the exception.

Since this is a synthetic error that's been generated specifically as an example, it is actually easy to find out the root cause. In the example code, we are explicitly raising an exception, which produces an error. This may not be the case in a real use case, where the exception could be generated in a different place than the actual error. Exceptions can be also originated in a different microservice within the same cluster.

After detecting the error, the objective should be to replicate it with a unit test in the microservice in order to generate the exception. This will allow us to replicate the conditions in a controlled environment.

If we run the tests for the Thoughts Backend code that's available in `Chapter10`, we will see errors because of this. Note that the logs are being displayed in failing tests:

```
$ docker-compose run test
...
___ ERROR at setup of test_get_non_existing_thought ___
-------- Captured log setup ---------
INFO flask.app:app.py:46 REQUEST POST /api/me/thoughts/
INFO flask.app:token_validation.py:66 Header successfully validated
ERROR flask.app:app.py:1761 Exception on /api/me/thoughts/ [POST]
Traceback (most recent call last):
  File "/opt/venv/lib/python3.6/site-packages/flask/app.py", line 1813, in
full_dispatch_request
    rv = self.dispatch_request()
  File "/opt/venv/lib/python3.6/site-packages/flask/app.py", line 1799, in
dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
  File "/opt/venv/lib/python3.6/site-packages/flask_restplus/api.py", line
325, in wrapper
    resp = resource(*args, **kwargs)
  File "/opt/venv/lib/python3.6/site-packages/flask/views.py", line 88, in
view
    return self.dispatch_request(*args, **kwargs)
  File "/opt/venv/lib/python3.6/site-packages/flask_restplus/resource.py",
line 44, in dispatch_request
    resp = meth(*args, **kwargs)
  File "/opt/venv/lib/python3.6/site-
packages/flask_restplus/marshalling.py", line 136, in wrapper
    resp = f(*args, **kwargs)
  File "/opt/code/thoughts_backend/api_namespace.py", line 80, in post
    raise Exception('Unexpected error!')
Exception: Unexpected error!
INFO flask.app:app.py:57 RESPONSE TIME 3 ms
INFO flask.app:app.py:60 RESPONSE STATUS 500
```

Once the error has been reproduced in unit tests, fixing it will often be trivial. Add a unit test to capture the set of conditions that trigger the error and then fix it. The new unit test will detect whether the error has been reintroduced on each automated build.

To fix the example code, remove the `raise` line of code. Then, things will work again.

Sometimes, the problem cannot be solved as it may be external. Maybe there's a problem in some of the rows in our database or maybe another service is returning incorrectly formatted data. In those cases, we can't completely avoid the root cause of the error. However, it's possible to capture the problem, do some remediation, and move from an unexpected error to an expected error.

Note that not every detected unexpected error is worth spending time on. Sometimes, uncaptured errors provide enough information on what the problem is, which is out of the realm of what the web service should handle; for example, there may be a network problem and the web service can't connect to the database. Use your judgment when you want to spend time on development.

Logging strategy

There's a problem when we're dealing with logs. What is the adequate level for a particular message? Is this a `WARNING` or an `ERROR`? Should this be an `INFO` statement?

Most of the log level descriptions use definitions such as *the program shows a potentially harmful situation* or *the program highlights the progress of the request*. These are vague and not very useful in a real-life environment. Instead, try to define each log level by relating them to the expected follow-up action. This helps provide clarity on what to do when a log of a particular level is found.

The following table shows some examples of the different levels and what action should be taken:

Log level	Action to take	Comments
`DEBUG`	Nothing.	Not tracked.
`INFO`	Nothing.	The `INFO` logs show generic information about the flow of the request to help track problems.
`WARNING`	Track number. Alert on raising levels.	The `WARNING` logs track errors that have been automatically fixed, such as retries to connect (but finally connecting) or fixable formatting errors in the database's data. A sudden increase may require investigation.
`ERROR`	Track number. Alert on raising levels. Review all.	The `ERROR` logs track errors that can't be fixed. A sudden increase may require immediate action so that this can be remediated.
`CRITICAL`	Immediate response.	A `CRITICAL` log indicates a catastrophic failure in the system. Even one will indicate that the system is not working and can't recover.

This is just a recommendation, but it sets clear expectations on how to respond. Depending on how your teams and your expected level of service work, you can adapt them to your use case.

Here, the hierarchy is very clear, and there's an acceptance that a certain number of `ERROR` logs will be generated. Not everything needs to be fixed immediately, but they should be noted and reviewed.

In real life, `ERROR` logs will be typically categorized as "we're doomed" or "meh." Development teams should actively either fix or remove "mehs" to reduce them as much as possible. That may include lowering the level of logs if they aren't covering actual errors. You want as few `ERROR` logs as possible, but all of them need to be meaningful.

Be pragmatic, though. Sometimes, errors can't be fixed straight away and time is best utilized in other tasks. However, teams should reserve time to reduce the number of errors that occur. Failing to do so will compromise the reliability of the system in the medium term.

`WARNING` logs are indications that something may not be working as smoothly as we expected, but there's no need to panic unless the numbers grow. `INFO` is just there to give us context if there's a problem, but otherwise should be ignored.

Avoid the temptation to produce an `ERROR` log when there's a request returning a 400 BAD REQUEST status code. Some developers will argue that if the customer sent a malformed request, it is actually an error. But this isn't something that you should care about if the request has been properly detected and returned. This is business as usual. If this behavior can lead to indicate something else, such as repeated attempts to send incorrect passwords, you can set a `WARNING` log. There's no point in generating `ERROR` logs when your system is behaving as expected.

As a rule of thumb, if a request is not returning some sort of 500 error (500, 502, 504, and so on), it should not generate an `ERROR` log. Remember the categorization of 400 errors as *you (customer) have a problem* versus 500 errors, which are categorized as *I have a problem*.

This is not absolute, though. For example, a spike in authentication errors that are normally 4XX errors may indicate that users cannot create logs due to a real internal problem.

With these definitions in mind, your development and operations teams will have a shared understanding that will help them take meaningful actions.

Expect to tweak the system and change some of the levels of the logs as your system matures.

Adding logs while developing

As we've already seen, properly configuring `pytest` will make any errors in tests display the captured logs.

This is an opportunity to check that the expected logs are being generated while a feature is in development. Any test that checks error conditions should also add its corresponding logs and check that they are being generated during the development of the feature.

You can check the logs as part of testing with a tool such as `pytest-catchlog` (`https://pypi.org/project/pytest-catchlog/`) to enforce that the proper logs are being produced.

Typically, though, just taking a bit of care and checking during development that logs are produced is enough for most cases. However, be sure that developers understand why it's useful to have logs while they're developing.

During development, `DEBUG` logs can be used to show extra information about the flow that will be too much for production. This may fill in the gaps between `INFO` logs and help us develop the habit of adding logs. A `DEBUG` log may be promoted to `INFO` if, during tests, it's discovered that it will be useful for tracking problems in production.

Potentially, `DEBUG` logs can be enabled in production in controlled cases to track some difficult problems, but be aware of the implications of having a large number of logs.

Be sensible with the information that's presented in `INFO` logs. In terms of the information that's displayed, avoid sensible data such as passwords, secret keys, credit card numbers, or personal information. This is the same for the number of logs.

Keep an eye on any size limitations and how quickly logs are being generated. Growing systems may have a log explosion while new features are being added, more requests are flowing through the system, and new workers are being added.

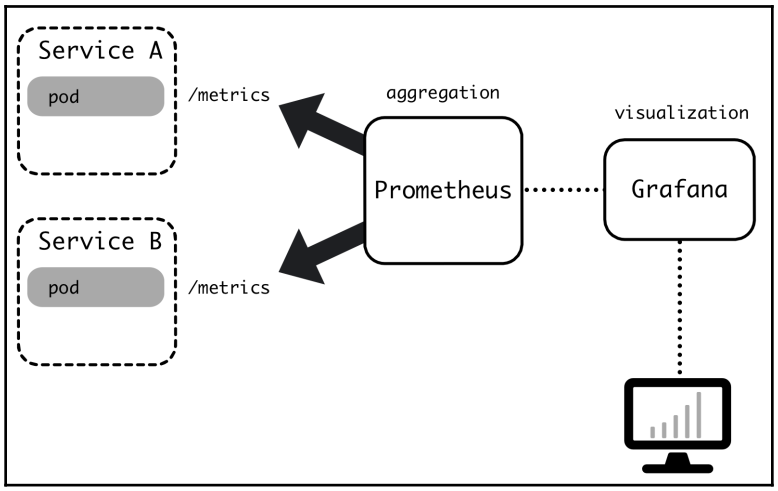
Also, double-check that the logs are being generated and captured correctly and that they work at all the different levels and environments. All of this configuration may take a bit of time, but you need to be very sure that you can capture unexpected errors in production and that all the plumbing is set correctly.

Let's take a look at the other key element when it comes to observability: metrics.

Setting up metrics

To set up metrics with Prometheus, we need to understand how the process works. Its key component is that each service that's measured has its own Prometheus client that keeps track of the metrics. The data in the Prometheus server will be available for a Grafana service that will plot the metrics.

The following diagram shows the general architecture:



The Prometheus server pulls information at regular intervals. This method of operation is very lightweight since registering metrics just updates the local memory of the service and scales well. On the other hand, it shows sampled data at certain times and doesn't register each individual event. This has certain implications in terms of storing and representing data and imposes limitations on the resolution of the data, especially for very low rates.

There are lots of available metrics exporters that will expose standard metrics in different systems, such as databases, hardware, HTTP servers, or storage. Check out the Prometheus documentation for more information: `https://prometheus.io/docs/instrumenting/exporters/`.

This means that each of our services needs to install a Prometheus client and expose its collected metrics in some way. We will use standard clients for Flask and Django.

Defining metrics for the Thoughts Backend

For Flask applications, we will use the `prometheus-flask-exporter` package (`https://github.com/rycus86/prometheus_flask_exporter`), which has been added to `requirements.txt`.

It gets activated in the `app.py` file (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter10/microservices/thoughts_backend/ThoughtsBackend/thoughts_backend/app.py#L95`) when the application is created.

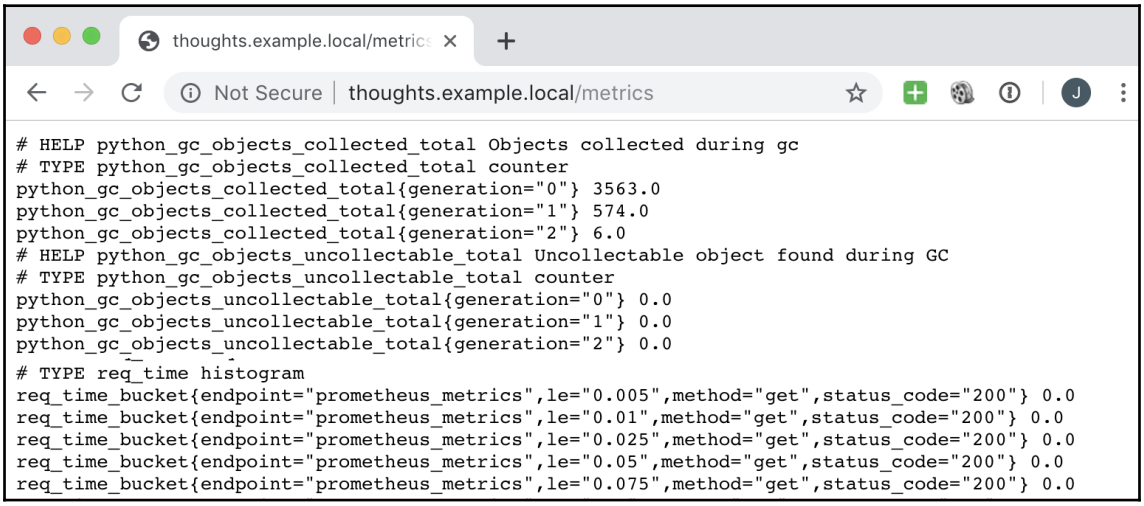
The `metrics` object is set up with no app, and is then instantiated in the `created_app` function:

```
from prometheus_flask_exporter import PrometheusMetrics

metrics = PrometheusMetrics(app=None)

def create_app(script=False):
    ...
    # Initialise metrics
    metrics.init_app(application)
```

This generates an endpoint in the `/metrics` service endpoint, that is, `http://thoughts.example.local/metrics`, which returns the data in Prometheus format. The Prometheus format is plain text, as shown in the following screenshot:



The default metrics that are captured by `prometheus-flask-exporter` are request calls based on the endpoint and the method (`flask_http_request_total`), as well as the time they took (`flask_http_request_duration_seconds`).

Adding custom metrics

We may want to add more specific metrics when it comes to application details. We also added some extra code at the end of the request so that we can store similar information to the metric that `prometheus-flask-exporter` allows us to.

In particular, we added this code to the `logging_after` function (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter10/microservices/thoughts_backend/ThoughtsBackend/thoughts_backend/app.py#L72`) using the lower-level `prometheus_client`.

This code creates `Counter` and `Histogram`:

```
from prometheus_client import Histogram, Counter

METRIC_REQUESTS = Counter('requests', 'Requests',
                          ['endpoint', 'method', 'status_code'])
METRIC_REQ_TIME = Histogram('req_time', 'Req time in ms',
                            ['endpoint', 'method', 'status_code'])

def logging_after(response):
    ...
    # Store metrics
    endpoint = request.endpoint
    method = request.method.lower()
    status_code = response.status_code
    METRIC_REQUESTS.labels(endpoint, method, status_code).inc()
    METRIC_REQ_TIME.labels(endpoint, method,
status_code).observe(time_in_ms)
```

Here, we've created two metrics: a counter called `requests` and a histogram called `req_time`. A histogram is a Prometheus implementation of measures and events that have a specific value, such as the request time (in our case).

The histogram stores the values in buckets, thereby making it possible for us to calculate quantiles. Quantiles are very useful to determine metrics such as the 95% value for times, such as the aggregate time, where 95% comes lower than it. This is much more useful than averages since outliers won't pull from the average.

There's another similar metric called summary. The differences are subtle, but generally, the metric we should use is a histogram. Check out the Prometheus documentation for more details (`https://prometheus.io/docs/practices/histograms/`).

The metrics are defined in `METRIC_REQUESTS` and `METRIC_REQ_TIME` by their name, their measurement, and the labels they define. Each label is an extra dimension of the metric, so you will be able to filter and aggregate by them. Here, we define the endpoint, the HTTP method, and the resulting HTTP status code.

For each request, the metric is updated. We need to set up the labels, the counter calls, that is, `.inc()`, and the histogram calls, that is, `.observe(time)`.

You can find the documentation for the Prometheus client at `https://github.com/prometheus/client_python`.

We can see the `request` and `req_time` metrics on the metrics page.

Setting up metrics for the Users Backend follows a similar pattern. The Users Backend is a similar Flask application, so we install `prometheus-flask-exporter` as well, but no custom metrics. You can access these metrics at `http://users.example.local/metrics`.

The next stage is to set up a Prometheus server so that we can collect the metrics and aggregate them properly.

Collecting the metrics

For this, we need to deploy the metrics using Kubernetes. We prepared a YAML file with everything set up already in the `Chapter10/kubernetes/prometheus.yaml` file.

This YAML file contains a deployment, a `ConfigMap`, which contains the configuration file, a service, and an Ingress. The service and Ingress are pretty standard, so we won't comment on them here.

The `ConfigMap` allows us to define a file:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: example
data:
  prometheus.yaml: |
    scrape_configs:
    - job_name: 'example'

      static_configs:
        - targets: ['thoughts-service', 'users-service',
                    'frontend-service']
```

Note how the `prometheus.yaml` file is generated after the `|` symbol. This is a minimal Prometheus configuration scraping from the `thoughts-service`, `users-service`, and `frontend-service` servers. As we know from the previous chapters, these names access the services and will connect to the pods that are serving the applications. They will automatically search for the `/metrics` path.

There is a small caveat here. From the point of view of Prometheus, everything behind the service is the same server. If you have more than one pod being served, the metrics that are being accessed by Prometheus will be load balanced and the metrics won't be correct.

This is fixable with a more complicated Prometheus setup whereby we install the Prometheus operator, but this is out of the scope of this book. However, this is highly recommended for a production system. In essence, it allows us to annotate each of the different deployments so that the Prometheus configuration is dynamically changed. This means we can access all the metrics endpoints exposed by the pods automatically once this has been set up. Prometheus Operator annotations make it very easy for us to add new elements to the metrics system.

Check out the following article if you want to learn how to do this: `https://sysdig.com/blog/kubernetes-monitoring-prometheus-operator-part3`.

The deployment creates a container from the public Prometheus image in `prom/prometheus`, as shown in the following code:

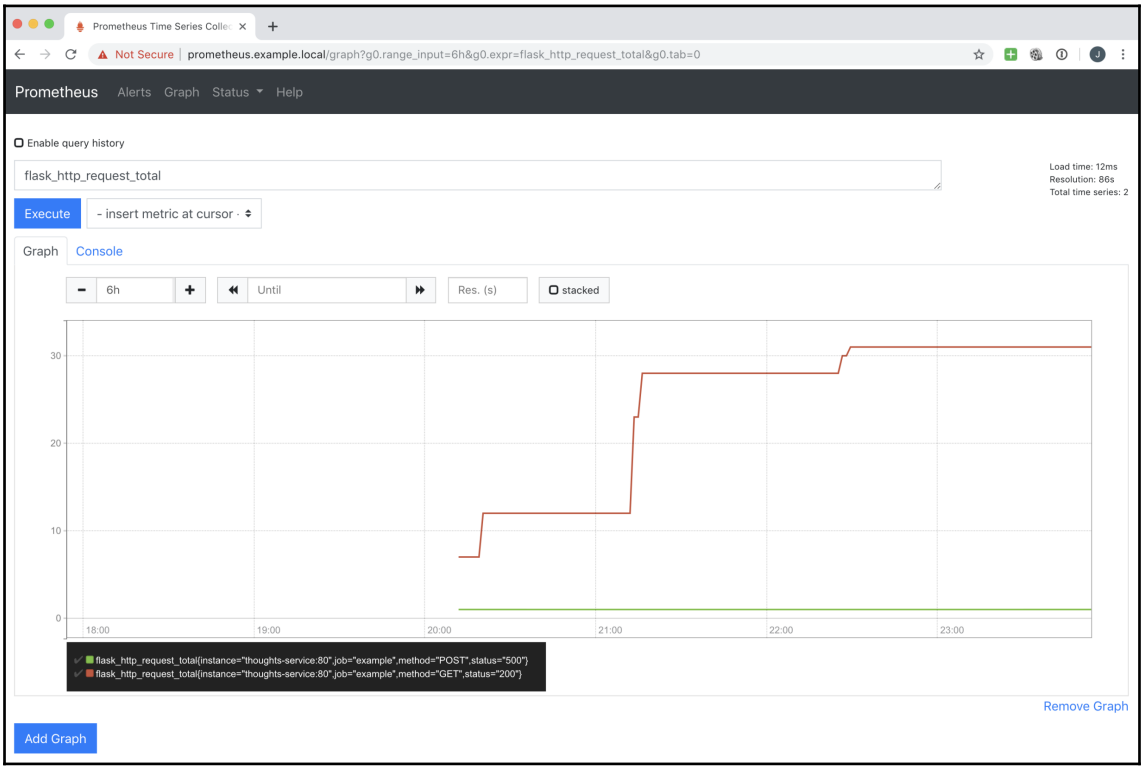
```
spec:
  containers:
  - name: prometheus
    image: prom/prometheus
    volumeMounts:
    - mountPath: /etc/prometheus/prometheus.yml
      subPath: prometheus.yaml
      name: volume-config
    ports:
    - containerPort: 9090
    volumes:
    - name: volume-config
      configMap:
        name: prometheus-config
```

It also mounts `ConfigMap` as a volume, and then as a file in `/etc/prometheus/prometheus.yml`. This starts the Prometheus server with that configuration. The container opens port `9090`, which is the default for Prometheus.

At this point, note how we delegated for the Prometheus container. This is one of the advantages of using Kubernetes: we can use standard available containers to add features to our cluster with minimal configuration. We don't even have to worry about the operating system or the packaging of the Prometheus container. This simplifies operations and allows us to standardize the tools we use.

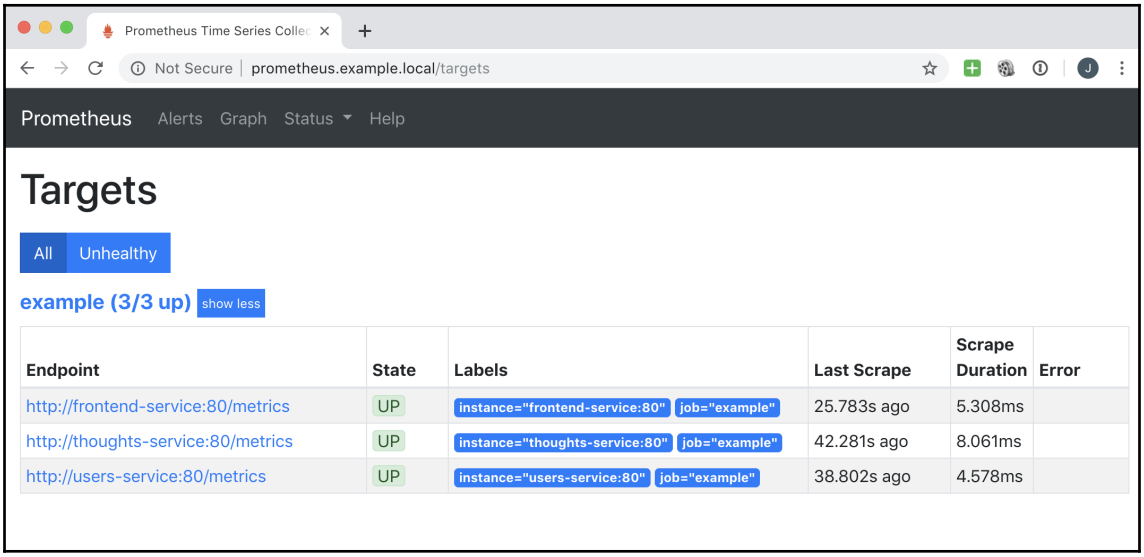
The deployed Prometheus server can be accessed at `http://prometheus.example.local/`, as described in the Ingress and service.

This displays a graphic interface that can be used to plot the graphs, as shown in the following screenshot:



The Expression search box will also autocomplete metrics, helping with the discovery process.

The interface also displays other elements from Prometheus that are interesting, such as the configuration or the statuses of the targets:



The graphs in this interface are usable, but we can set up more complicated and useful dashboards through Grafana. Let's see how this setup works.

Plotting graphs and dashboards

The required Kubernetes configuration, `grafana.yaml`, is available in this book's GitHub repository in the `Chapter10/kubernetes/metrics` directory. Just like we did with Prometheus, we used a single file to configure Grafana.

We won't show the Ingress and service for the same reason we explained previously. The deployment is simple, but we mount two volumes instead of one, as shown in the following code:

```
spec:
  containers:
    - name: grafana
      image: grafana/grafana
      volumeMounts:
```

```
        - mountPath: /etc/grafana/provisioning
                     /datasources/prometheus.yaml
          subPath: prometheus.yaml
          name: volume-config
        - mountPath: /etc/grafana/provisioning/dashboards
          name: volume-dashboard
      ports:
        - containerPort: 3000
  volumes:
    - name: volume-config
      configMap:
        name: grafana-config
    - name: volume-dashboard
      configMap:
        name: grafana-dashboard
```

The `volume-config` volume shares a single file that configures Grafana. The `volume-dashboard` volume adds a dashboard. The latter mounts a directory that contains two files. Both mounts are in the default location that Grafana expects for configuration files.

The `volume-config` volume sets up the data source in the place where Grafana will receive the data to plot:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-config
  namespace: example
data:
  prometheus.yaml: |
      apiVersion: 1

      datasources:
      - name: Prometheus
        type: prometheus
        url: http://prometheus-service
        access: proxy
        isDefault: true
```

The data comes from `http://prometheus-service` and points to the Prometheus service we configured previously.

`volume-dashboard` defines two files, `dashboard.yaml` and `dashboard.json`:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-dashboard
  namespace: example
data:
  dashboard.yaml: |
    apiVersion: 1

    providers:
    - name: 'Example'
      orgId: 1
      folder: ''
      type: file
      editable: true
      options:
        path: /etc/grafana/provisioning/dashboards
  dashboard.json: |-
    <JSON FILE>
```

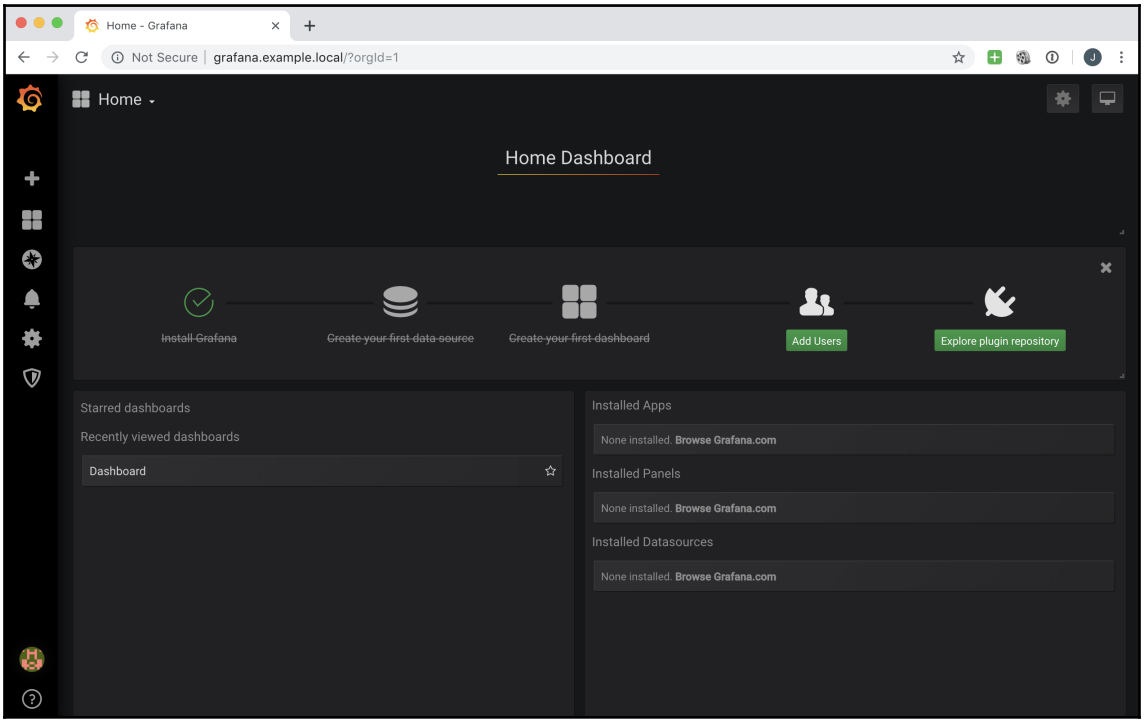
`dashboard.yaml` is a simple file that points to the directory where we can find JSON files describing the available dashboards for the system. We point to the same directory to mount everything with a single volume.

`dashboard.json` is redacted here to save space; check out this book's GitHub repository for the data.

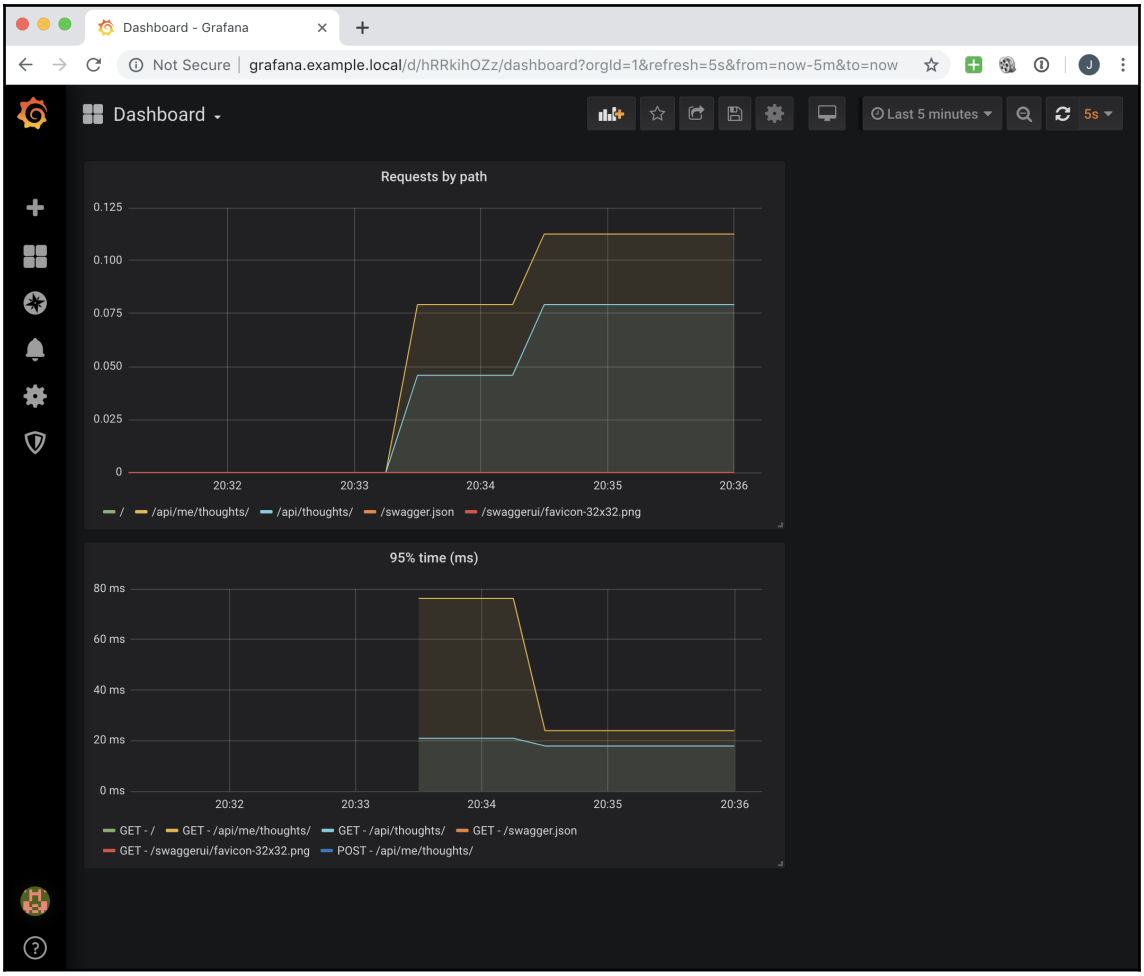
`dashboard.json` describes a dashboard in JSON format. This file can be automatically generated through the Grafana UI. Adding more `.json` files will create new dashboards.

Grafana UI

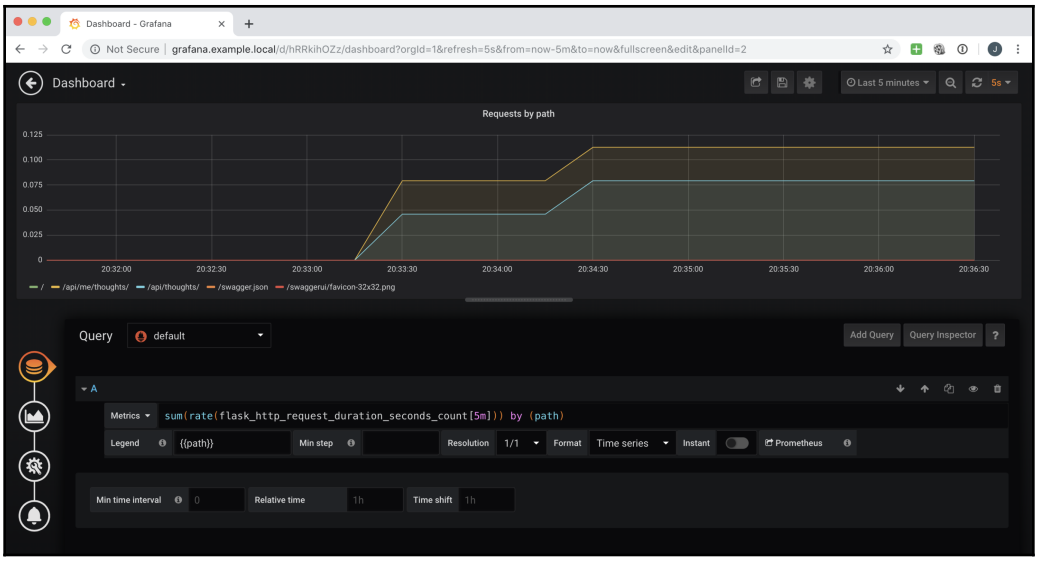
By accessing `http://grafana.example.local` and using your login/password details, that is, `admin/admin` (the default values), you can access the Grafana UI:



From there, you can check the dashboard, which can be found in the left central column:



This captures the calls to Flask, both in terms of numbers and in 95^{th} percentile time. Each individual graph can be edited so that we can see the recipe that produces it:



The icons on the left allow us to change the queries that are running in the system, change the visualization (units, colors, bars or lines, kind of scale to plot, and so on), add general information such as name, and create alerts.

The Grafana UI allows us to experiment and so is highly interactive. Take some time to try out the different options and learn how to present the data.

The **Query** section allows us to add and display metrics from Prometheus. Note the Prometheus logo near **default**, which is the data source.

Each of the queries has a **Metrics** section that extracts data from Prometheus.

Querying Prometheus

Prometheus has its own query language called PromQL. The language is very powerful, but it presents some peculiarities.

The Grafana UI helps by autocompleting the query, which makes it easy for us to search for metric names. You can experiment directly in the dashboard, but there's a page on Grafana called **Explore** that allows you to make queries out of any dashboard and has some nice tips, including basic elements. This is denoted by a compass icon in the left sidebar.

The first thing to keep in mind is understanding the Prometheus metrics. Given its sampling approach, most of them are monotonically increasing. This means that plotting the metrics will show a line going up and up.

To get the rate at which the value changes over a period of time, you need to use `rate`:

```
rate(flask_http_request_duration_seconds_count[5m])
```

This generates the requests per second, on average, with a moving window of `5` minutes. The rate can be further aggregated using `sum` and `by`:

```
sum(rate(flask_http_request_duration_seconds_count[5m])) by (path)
```

To calculate the times, you can use `avg` instead. You can also group by more than one label:

```
avg(rate(flask_http_request_duration_seconds_bucket[5m])) by (method, path)
```

However, you can also set up quantiles, just like we can in graphs. We multiply by 100 to get the time in milliseconds instead of seconds and group by `method` and `path`. Now, `le` is a special tag that's created automatically and divides the data into multiple buckets. The `histogram_quantile` function uses this to calculate the quantiles:

```
histogram_quantile(0.95,
sum(rate(flask_http_request_duration_seconds_bucket[5m])) by (method, path,
le)) * 1000
```

Metrics can be filtered so that only specific labels are displayed. They can also be used for different functions, such as division, multiplication, and so on.

Prometheus queries can be a bit long and complicated when we're trying to display the result of several metrics, such as the percentage of successful requests over the total. Be sure to test that the result is what you expect it to be and allocate time to tweak the requests, later.

Be sure to check out the Prometheus documentation if you want to find out more: `https://prometheus.io/docs/prometheus/latest/querying/basics/`.

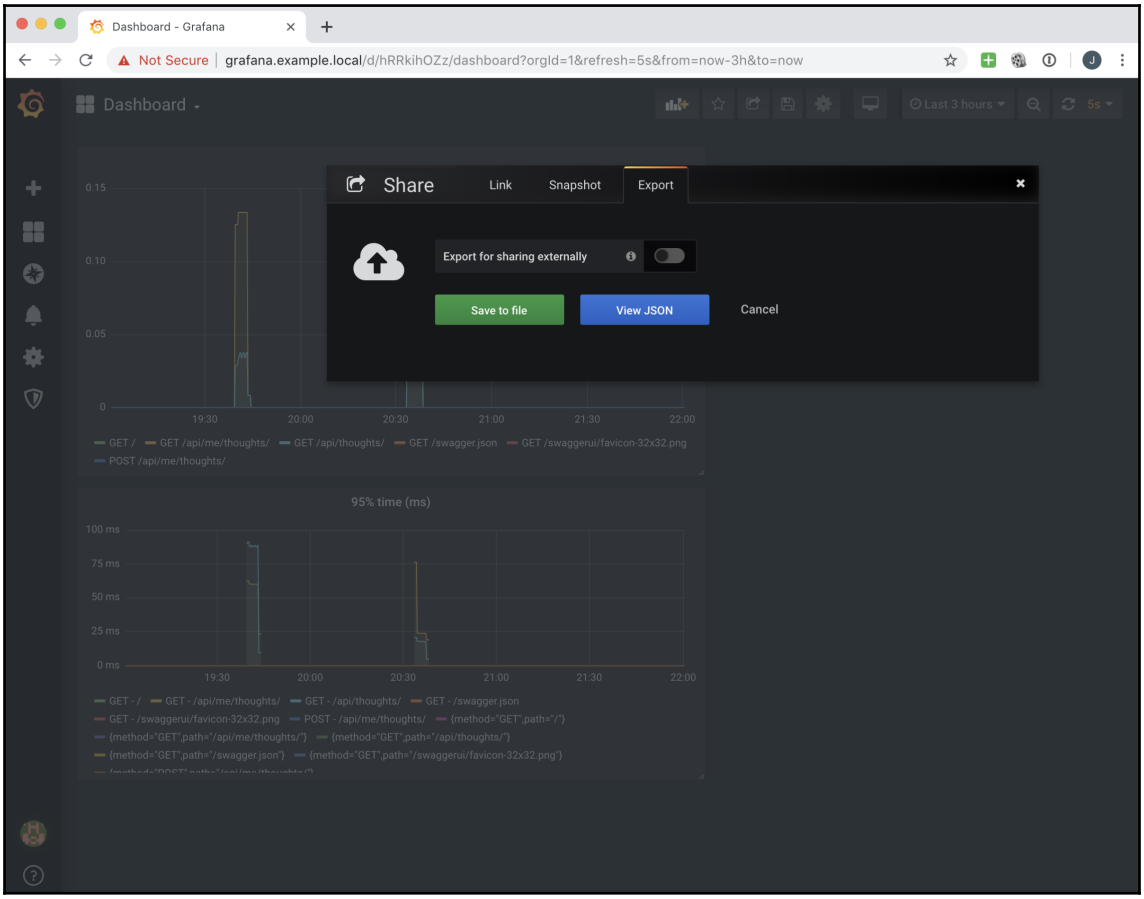
Updating dashboards

Dashboards can be interactively changed and saved, but in our Kubernetes configuration, we set up the volumes that contain the files as non-persistent. Due to this, restarting Grafana will discard any changes and reapply the defined configuration in `volume-dashboard` in the `Chapter10/kubernetes/metrics/grafana.yaml` file.

This is actually a good thing since we apply the same GitOps principles to store the full configuration in the repository under Git source control.

However, as you can see, the full JSON description of the dashboard contained in the `grafana.yaml` file is very long, given the number of parameters and the difficulty to change them manually.

The best approach is to change the dashboard interactively and then export it into a JSON file with the **Share file** button at the top of the menu. Then, the JSON file can be added to the configuration:



The Grafana pod can then be redeployed and will contain the saved changes in the dashboard. The Kubernetes configuration can then be updated in Git through the usual process.

Be sure to explore all the possibilities for dashboards, including the option to set up variables so that you can use the same dashboard to monitor different applications or environments and the different kinds of visualization tools. See the full Grafana documentation for more information: `https://grafana.com/docs/reference/`.

Having metrics available allows us to use them to proactively understand the system and anticipate any problems.

Being proactive

Metrics show an aggregated point of view for the status of the whole cluster. They allow us to detect trending problems, but it's difficult to find a single spurious error.

Don't underestimate them, though. They are critical for successful monitoring because they tell us whether the system is healthy. In some companies, the most critical metrics are prominently displayed in screens on the wall so that the operations team can see them at all times and quickly react.

Finding the proper balance for metrics in a system is not a straightforward task and will require time and trial and error. There are four metrics for online services that are always important, though. These are as follows:

- **Latency**: How many milliseconds the system takes to respond to a request.

Depending on the times, a different time unit, such as seconds or microseconds, can be used. From my experience, milliseconds is adequate since most of the requests in a web application system should take between 50 ms and 1 second to respond. Here, a system that takes 50 ms is too slow and one that takes 1 second is a very performant one.

- **Traffic**: The number of requests flowing through the system per unit of time, that is, requests per second or per minute.
- **Errors**: The percentage of requests received that return an error.
- **Saturation**: Whether the capacity of the cluster has enough headroom. This includes elements such as hard drive space, memory, and so on. For example, there is 20% available RAM memory.

To measure saturation, remember to install the available exporters that will collect most of the hardware information (memory, hard disk space, and so on) automatically. If you use a cloud provider, normally, they expose their own set of related metrics as well, for example, CloudWatch for AWS.

These metrics can be found in the Google SRE Book as *the Four Golden Signals* and are recognized as the most important high-level elements for successful monitoring.

Alerting

When problems arise in metrics, an automatic alert should be generated. Prometheus has an included alert system that will trigger when a defined metric fulfills the defined condition.

Check out the Prometheus documentation on alerting for more information: `https://prometheus.io/docs/alerting/overview/`.

Prometheus' Alertmanager can perform certain actions, such as sending emails to be notified based on rules. This system can be connected to an integrated incident solution such as OpsGenie (`https://www.opsgenie.com`) in order to generate all kinds of alerts and notifications, such as emails, SMS, calls, and so on.

Logs can also be used to create alerts. There are certain tools that allow us to create an entry when an `ERROR` is raised, such as **Sentry**. This allows us to detect problems and proactively remediate them, even if the health of the cluster hasn't been compromised.

Some commercial tools that handle logs, such as Loggly, allow us to derive metrics from the logs themselves, plotting graphs either based on the kind of log or extracting values from them and using them as values. While not as complete as a system such as Prometheus, they can monitor some values. They also allow us to notify if thresholds are reached.

The monitoring space is full of products, both free and paid, that can help us to handle this. While it's possible to create a completely in-house monitoring system, being able to analyze whether commercial cloud tools will be of help is crucial. The level of features and their integration with useful tools such as external alerting systems will be difficult to replicate and maintain.

Alerting is also an ongoing process. Some elements will be discovered down the line and new alerts will have to be created. Be sure to invest time so that everything works as expected. Logs and metrics will be used while the system is unhealthy, and in those moments, time is critical. You don't want to be guessing about logs because the host parameter hasn't been configured correctly.

Being prepared

In the same way that a backup is not useful unless the recovery process has been tested and is working, be proactive when checking that the monitoring system is producing useful information.

In particular, try to standardize the logs so that there's a good expectation about what information to include and how it's structured. Different systems may produce different logs, but it's good to make all the microservices log in the same format. Double-check that any parameters, such as client references or hosts, are being logged correctly.

The same applies to metrics. Having a set of metrics and dashboards that everyone understands will save a lot of time when you're tracking a problem.

Summary

In this chapter, we learned how to work with logs and metrics, as well as how to set up logs and send them to a centralized container using the `syslog` protocol. We described how to add logs to different applications, how to include a request ID, and how to raise custom logs from the different microservices. Then, we learned how to define a strategy to ensure that the logs are useful in production.

We also described how to set up standard and custom Prometheus metrics in all the microservices. We started a Prometheus server and configured it so that it collects metrics from our services. We started a Grafana service so that we can plot the metrics and created dashboards so that we can display the status of the cluster and the different services that are running.

Then, we introduced you to the alert system in Prometheus and how it can be used so that it notifies us of problems. Remember that there are commercial services to help you with logs, metrics, and alerts. Analyze your options as they can save you a lot of time and money in terms of maintenance costs.

In the next chapter, we will learn how to manage changes and dependencies that affect several microservices and how to handle configurations and secrets.

Questions

1. What is the observability of a system?
2. What are the different severity levels that are available in logs?
3. What are metrics used for?
4. Why do you need to add a request ID to logs?
5. What are the available kinds of metrics in Prometheus?
6. What is the 75th percentile in a metric and how does it differ from the average?
7. What are the four golden signals?

Further reading

You can learn more about monitoring with different tools and techniques while using Docker by reading *Monitoring Docker* (`https://www.packtpub.com/virtualization-and-cloud/monitoring-docker`). To find out more about Prometheus and Grafana, including how to set up alerts, please read *Hands-On Infrastructure Monitoring with Prometheus* (`https://www.packtpub.com/virtualization-and-cloud/hands-infrastructure-monitoring-prometheus`).

Monitoring is only the starting point of successfully running services reliably. To find out how to successfully improve your operations, check out *Real-World SRE* (`https://www.packtpub.com/web-development/real-world-sre`).

11
Handling Change, Dependencies, and Secrets in the System

In this chapter, we will describe different elements that interact with multiple microservices.

We will look at strategies on how to make services describe their version so that dependent microservices can discover them and be sure that they have the proper dependencies already deployed. This will allow us to define a deploying order in dependent services and will stop deployment of a service if not all dependencies are ready.

This chapter describes how to define configuration parameters that are cluster-wide, so they can be shared across multiple microservices and managed in a single place, using Kubernetes ConfigMap. We will also learn how to deal with configuration parameters that are secrets—like encryption keys—that should not be accessible to most people on the team.

The following topics will be covered in this chapter:

- Understanding shared configuration across microservices
- Handling Kubernetes secrets
- Defining a new feature affecting multiple services
- Dealing with service dependencies

By the end of this chapter, you'll know how to prepare dependent services for safe deployment and how to include secrets in your microservices that won't be accessible outside the deployment they're intended for.

Technical requirements

The code is available on GitHub at the following URL: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter11`. Note that the code is an extension of the code in `Chapter10`, with extra elements as described in this chapter. The structure is the same a subdirectory called `microservices` with the code, and another one called `kubernetes` with the Kubernetes configuration files.

To install the cluster, you need to build each individual microservice with the following commands:

```
$ cd Chapter11/microservices/
$ cd rsyslog
$ docker-compose build
...
$ cd frontend
$ ./build-test.sh
...
$ cd thoughts_backend
$./build-test.sh
...
$ cd users_backend
$ ./build-test.sh
...
```

This will build the required services.

Note that we use the `build-test.sh` script. We will explain how it works in this chapter.

And then, create the `namespace` example and start the Kubernetes cluster with the configuration found in the `Chapter11/kubernetes` subdirectory:

```
$ cd Chapter11/kubernetes
$ kubectl create namespace example
$ kubectl apply --recursive -f .
...
```

This deploys the microservices to the cluster.

The code included in `Chapter11` has some issues and **won't** deploy correctly until it is fixed. This is the expected behavior. During the chapter, we will explain the two problems: the secrets not getting configured, and the dependency for Frontend not getting fulfilled, stopping it from starting.

Keep reading the chapter to find the problems described. The solution is proposed as an assessment.

To be able to access the different services, you need to update your `/etc/hosts` file to include the following lines:

```
127.0.0.1 thoughts.example.local
127.0.0.1 users.example.local
127.0.0.1 frontend.example.local
```

With that, you will be able to access services for this chapter.

Understanding shared configurations across microservices

Some configurations may be common to several microservices. In our example, we are duplicating the same values for the database connection. Instead of repeating the values on each of the deployment files, we can use ConfigMap and share it across the different deployments.

We've seen how to add ConfigMap to include files in `Chapter 10`, *Monitoring Logs and Metrics*, under the *Setting up metrics* section. It was used for a single service, though.

A ConfigMap is a group of key/value elements. They can be added as environment variables or as files. In the next section, we will add a general configuration file that includes all the shared variables in the cluster.

Adding the ConfigMap file

The `configuration.yaml` file contains the common configuration of the system. It is available in the `Chapter11/kubernetes` subdirectory:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: shared-config
  namespace: example
data:
  DATABASE_ENGINE: POSTGRES
  POSTGRES_USER: postgres
  POSTGRES_HOST: "127.0.0.1"
  POSTGRES_PORT: "5432"
  THOUGHTS_BACKEND_URL: http://thoughts-service
  USER_BACKEND_URL: http://users-service
```

The variables related to the database, `DATABASE_ENGINE`, `POSTGRES_USER`, `POSTGRES_HOST`, and `POSTGRES_PORT`, are shared across the Thoughts Backend and Users Backend.

The `POSTGRES_PASSWORD` variable is a secret. We will describe this later in this chapter in the *Handling Kubernetes secrets* section.

The `THOUGHTS_BACKEND_URL` and `USER_BACKEND_URL` variables are used in the Frontend service. They are common across the cluster, though. Any service that wants to connect to the Thoughts Backend should use the same URL as described in `THOUGHTS_BACKEND_URL`.

Even though it's only used in a single service, Frontend, so far, it fits the description of a variable that's system-wide and should be included in the general configuration.

One of the advantages of having a shared repository for variables is to consolidate them.

While creating multiple services and developing them independently, it is quite common to end up using the same information, but in two slightly different ways. Teams developing independently won't be able to share information perfectly, and this kind of mismatch will happen.

For example, one service can describe an endpoint as `URL=http://service/api`, and another service using the same endpoint will describe it as `HOST=service PATH=/api`. The code of each service handles the configuration differently, though they connect to the same endpoint. This makes it more difficult to change the endpoint in a unified way, as it needs to be changed in two or more places, in two ways.

A shared place is a good way to first detect these problems, as they normally go undetected if each service keeps its own independent configuration, and then to adapt the services to use the same variable, reducing the complexity of the configuration.

The name of ConfigMap in our example is `shared-config` as defined in the metadata and, like any other Kubernetes object, it can be managed through `kubectl` commands.

Using kubectl commands

The ConfigMap information can be checked with the usual set of `kubectl` commands. This allows us to discover the defined ConfigMap instances in the cluster:

```
$ kubectl get configmap -n example shared-config
NAME            DATA AGE
shared-config   6    46m
```

Note how the number of keys, or variables, that ConfigMap contains is displayed; here, it is `6`. To see the content of ConfigMap, use `describe`:

```
$ kubectl describe configmap -n example shared-config
Name: shared-config
Namespace: example
Labels: <none>
Annotations: kubectl.kubernetes.io/last-applied-configuration:
{"apiVersion":"v1","data":{"DATABASE_ENGINE":"POSTGRES","POSTGRES_HOST":"12
7.0.0.1","POSTGRES_PORT":"5432","POSTGRES_USER":"postgres","THO...
```

```
Data
====
POSTGRES_HOST:
----
127.0.0.1
POSTGRES_PORT:
----
5432
POSTGRES_USER:
----
postgres
THOUGHTS_BACKEND_URL:
----
http://thoughts-service
USER_BACKEND_URL:
----
http://users-service
DATABASE_ENGINE:
----
POSTGRES
```

If you need to change ConfigMap, you can use the `kubectl edit` command, or, even better, change the `configuration.yaml` file and reapply it using the following command:

```
$ kubectl apply -f kubernetes/configuration.yaml
```

This will overwrite all the values.

The configuration won't be applied automatically to the Kubernetes cluster. You'll need to redeploy the pods affected by the changes. The easiest way is to delete the affected pods and allow the deployment to recreate them.

On the other hand, if Flux is configured, it will redeploy the dependent pods automatically. Keep in mind that a change in ConfigMap (referenced in all pods) will trigger a redeploy on all pods in that situation.

We will now see how to add ConfigMap to the deployments.

Adding ConfigMap to the deployment

Once ConfigMap is in place, it can be used to share its variables with different deployments, maintaining a central location where to change the variables and avoid duplication.

Let's see how each of the deployments for the microservices (Thoughts Backend, Users Backend, and Frontend) makes use of the `shared-config` ConfigMap.

Thoughts Backend ConfigMap configuration

The Thoughts Backend deployment is defined as follows:

```
spec:
    containers:
        - name: thoughts-backend-service
          image: thoughts_server:v1.5
          imagePullPolicy: Never
          ports:
              - containerPort: 8000
          envFrom:
              - configMapRef:
                    name: shared-config
          env:
              - name: POSTGRES_DB
                value: thoughts
          ...
```

The full `shared-config` ConfigMap will be injected into the pod. Note that this includes the `THOUGHTS_BACKEND_URL` and `USER_BACKEND_URL` environment variables that previously were not available in the pod. More environment variables can be added. Here, we left `POSTGRES_DB` instead of adding it to the ConfigMap.

We can use `exec` in the pod to confirm it.

Note that to be able to connect the secret, it should be properly configured. Refer to the *Handling Kubernetes secrets* section.

To check inside the container, retrieve the pod name and use `exec` in it, as shown in the following commands:

```
$ kubectl get pods -n example
NAME                                READY STATUS  RESTARTS AGE
thoughts-backend-5c8484d74d-ql8hv 2/2   Running 0        17m
...
$ kubectl exec -it thoughts-backend-5c8484d74d-ql8hv -n example /bin/sh
Defaulting container name to thoughts-backend-service.
/opt/code $ env | grep POSTGRES
DATABASE_ENGINE=POSTGRESQL
POSTGRES_HOST=127.0.0.1
POSTGRES_USER=postgres
POSTGRES_PORT=5432
POSTGRES_DB=thoughts
/opt/code $ env | grep URL
THOUGHTS_BACKEND_URL=http://thoughts-service
USER_BACKEND_URL=http://users-service
```

The `env` command returns all the environment variables, but there are a lot of them added automatically by Kubernetes.

Users Backend ConfigMap configuration

The Users Backend configuration is similar to the previous type of configuration we just saw:

```
spec:
    containers:
        - name: users-backend-service
          image: users_server:v2.3
          imagePullPolicy: Never
          ports:
              - containerPort: 8000
          envFrom:
              - configMapRef:
                    name: shared-config
          env:
              - name: POSTGRES_DB
                value: thoughts
          ...
```

The value of `POSTGRES_DB` is the same as in the Thoughts Backend, but we left it here to show how you can add more environment variables.

Frontend ConfigMap configuration

The Frontend configuration only uses ConfigMap, as no extra environment variables are required:

```
spec:
    containers:
        - name: frontend-service
          image: thoughts_frontend:v3.7
          imagePullPolicy: Never
          ports:
              - containerPort: 8000
          envFrom:
              - configMapRef:
                    name: shared-config
```

The Frontend pods will also now include the information about the connection to the database, something that it doesn't require. This is fine for most of the configuration parameters.

You can also use multiple ConfigMaps to describe different groups of configurations, if necessary. It is simpler to handle them in a big bucket with all the configuration parameters, though. This will help to catch duplicated parameters and ensure that you have all the required parameters in all microservices.

However, some configuration parameters have to be handled with more care as they'll be sensitive. For example, we left out from the `shared-config` ConfigMap the `POSTGRES_PASSWORD` variable. This allows us to log into the database, and it should not be stored on any file with other parameters, so as to avoid accidental exposure.

To deal with this kind of information, we can use Kubernetes secrets.

Handling Kubernetes secrets

Secrets are a special kind of configuration. They need to be protected from being read by the other microservices that are using them. They are typically sensitive data, such as private keys, encryption keys, and passwords.

Remember that reading a secret is a valid operation. After all, they need to be used. What differentiates a secret from other configuration parameters is that they need to be protected so only the authorized sources are able to read them.

The secrets should be injected by the environment. This requires the code to be able to retrieve the configuration secrets and use the proper one for the current environment. It also avoids storing the secret inside the code.

Remember *never* to commit production secrets in your Git repositories. The Git tree means that, even if it's deleted, the secret is retrievable. This includes the GitOps environment.

Also, use different secrets for different environments. The production secrets require more care than the ones in test environments.

In our Kubernetes configuration, the authorized sources are the microservices using them, as well as administrators of the system, accessing through `kubectl`.

Let's see how to manage these secrets.

Storing secrets in Kubernetes

Kubernetes deals with secrets as a particular kind of ConfigMap values. They can be defined in the system and then applied in the same say a ConfigMap will be. The difference with the general ConfigMaps is that the information is protected internally. While they can be accessed through `kubectl`, they are protected against accidental exposure.

A secret can be created in the cluster through `kubectl` commands. They should *not* be created through files and GitOps or Flux, but manually instead. This avoids storing the secrets under the GitOps repo.

The pods that require the secret to operate will indicate so in their deployment file. This is safe to store under GitOps source control, as it doesn't store the secret but only the reference to the secret. When the pod gets deployed, it will use the proper reference and decode secret.

Logging into the pod will grant you access to the secret. This is normal, since, inside the pod, the application needs to read its value. Granting access to execute commands in the pod will grant them access to the secrets inside, so keep it in mind. You can read Kubernetes documentation about the best practices of the secrets to understand and adjust depending on your requirements (`https://kubernetes.io/docs/concepts/configuration/secret/#best-practices`).

Now that we know how to handle them, let's see how to create these secrets.

Creating the secrets

Let's create the secrets in Kubernetes. We will store the following secrets:

- The PostgreSQL password
- The public and private keys to sign and validate requests

We will store them inside the same Kubernetes secret that can have multiple keys. The following commands show how to generate a pair of keys:

```
$ openssl genrsa -out private_key.pem 2048
Generating RSA private key, 2048 bit long modulus
........+++
..................+++
e is 65537 (0x10001)
$ openssl rsa -in private_key.pem -outform PEM -pubout -out public_key.pub
writing RSA key
$ ls
private_key.pem public_key.pub
```

These keys are unique to you. We will use them to replace the stored example keys in the previous chapters.

Storing the secrets in the cluster

Store the secrets in the cluster, under the `thoughts-secrets` secret. Remember to store it in the `example` namespace:

```
$ kubectl create secret generic thoughts-secrets --from-literal=postgres-
password=somepassword --from-file=private_key.pem --from-
file=public_key.pub -n example
```

You can list the secrets in the namespace:

```
$ kubectl get secrets -n example
NAME              TYPE   DATA AGE
thoughts-secrets Opaque 3    41s
```

And you can describe the secrets for more info:

```
$ kubectl describe secret thoughts-secrets -n example
Name: thoughts-secrets
Namespace: default
Labels: <none>
Annotations: <none>
```

```
Type: Opaque

Data
====
postgres-password: 12 bytes
private_key.pem: 1831 bytes
public_key.pub: 408 bytes
```

You can get the content of a secret, but the data gets retrieved encoded in Base64.

Base64 is an encoding scheme that allows you to transform binary data into text and vice versa. It is widely used. This allows you to store any binary secret, not only text. It also means that the secrets are not displayed in plain text when retrieved, adding a small layer of protection in cases such as unintentional display in screens.

To obtain the secret, use the usual `kubectl get` command as shown here. We use the `base64` command to decode it:

```
$ kubectl get secret thoughts-secrets -o yaml -n example
apiVersion: v1
data:
  postgres-password: c29tZXBhc3N3b3Jk
  private_key.pem: ...
  public_key.pub: ...
$ echo c29tZXBhc3N3b3Jk | base64 --decode
somepassword
```

In the same way, if you edit a secret to update it, the input should be encoded in Base64.

Secret deployment configuration

We need to configure the secret usage in the deployment configuration, so the secret is available in the required pod. For example, in the User Backend `deployment.yaml` config file, we have the following code:

```
spec:
    containers:
    - name: users-backend-service
      ...
      env:
      ...
      - name: POSTGRES_PASSWORD
        valueFrom:
          secretKeyRef:
            name: thoughts-secrets
```

```
            key: postgres-password
        volumeMounts:
        - name: sign-keys
          mountPath: "/opt/keys/"

    volumes:
    - name: sign-keys
      secret:
        secretName: thoughts-secrets
        items:
        - key: public_key.pub
          path: public_key.pub
        - key: private_key.pem
          path: private_key.pem
```

We create the `POSTGRES_PASSWORD` environment variable that comes directly from the secret. We also create a volume called `sign-keys` that contains two keys as files, `public_key.pub` and `private_key.pem`. It mounts in the `/opt/keys/` path.

In a similar way, the `deployment.yaml` file for the Thoughts Backend includes secrets, but only the PostgreSQL password and `public_key.pub`. Note that the private key is not added, as the Thoughts Backend doesn't require it, and it's not available.

For the Frontend, only the public key is required. Now, let's establish how to retrieve the secrets.

Retrieving the secrets by the applications

For the `POSTGRES_PASSWORD` environment variable, we don't need to change anything. It was already an environment variable and the code was extracting it from there.

But for the secrets stored as files, we need to retrieve them from the proper location. The secrets stored as files are the key to signing the authentication headers. The public file is required in all the microservices, and the private key only in the Users Backend.

Now, let's take a look at the `config.py` file for the Users Backend:

```
import os
PRIVATE_KEY = ...
PUBLIC_KEY = ...

PUBLIC_KEY_PATH = '/opt/keys/public_key.pub'
PRIVATE_KEY_PATH = '/opt/keys/private_key.pem'

if os.path.isfile(PUBLIC_KEY_PATH):
```

```
        with open(PUBLIC_KEY_PATH) as fp:
            PUBLIC_KEY = fp.read()

if os.path.isfile(PRIVATE_KEY_PATH):
    with open(PRIVATE_KEY_PATH) as fp:
        PRIVATE_KEY = fp.read()
```

The current keys are still there as default values. They will be used for unit tests when the secret files are not mounted.

It is worth saying it again, but please *do not* use any of these keys. These are for running tests only and available to anyone that has access to this book.

If the files in the `/opt/keys/` path are present, they'll be read, and the content will be stored in the proper constant. The Users Backend requires both the public and private keys.

In the Thoughts Backend `config.py` file, we only retrieve the public key, as seen in the following code:

```
import os
PUBLIC_KEY = ...

PUBLIC_KEY_PATH = '/opt/keys/public_key.pub'

if os.path.isfile(PUBLIC_KEY_PATH):
    with open(PUBLIC_KEY_PATH) as fp:
        PUBLIC_KEY = fp.read()
```

The Frontend service adds the public key in the `settings.py` file:

```
TOKENS_PUBLIC_KEY = ...

PUBLIC_KEY_PATH = '/opt/keys/public_key.pub'

if os.path.isfile(PUBLIC_KEY_PATH):
    with open(PUBLIC_KEY_PATH) as fp:
        TOKENS_PUBLIC_KEY = fp.read()
```

This configuration makes the secret available for the applications and closes the loop for the secret values. Now, the microservices cluster uses the signing key from a secret value, which is a safe way of storing sensible data.

Defining a new feature affecting multiple services

We talked about change requests within the realm of a single microservice. But what if we need to deploy a feature that works within two or more microservices?

These kinds of features should be relatively rare and are one of the main causes of overhead in microservices compared with the monolith approach. In a monolith, this case is simply not possible as everything is contained within the walls of the monolith.

In a microservice architecture, meanwhile, this is a complex change. This involves at least two independent features on each involved microservice that resides in two different repos. It is likely that the repos will be developed by two different teams, or at least different people will be responsible for each of the features.

Deploying one change at a time

To ensure that the features can be deployed smoothly, one at a time, they need to keep backward compatibility. This means that you need to be able to live in an intermediate stage when service A has been deployed, but not service B. Each change in the microservices needs to be as small as possible to minimize risks, and they should be introduced one change at a time.

Why don't we deploy them all simultaneously? Because releasing two microservices at the same time is dangerous. To start with, deployments are not instantaneous, so there will be moments where out-of-date services will either send or receive calls that the system is not prepared to handle. That will create errors that may affect your customers.

But there's a chance of a situation occurring where one of the microservices is incorrect and needs to be rolled back. Then, the system is left in an inconsistent state. The dependent microservice needs to be rolled back as well. This, in itself, is problematic, but it can make things worse when, during the debugging of this problem, both microservices are stuck and cannot be updated until the problem gets fixed.

In a healthy microservice environment, there will be deployments happening quite often. Having to stop the pipeline for a microservice because another service requires work is a bad position to be in, and it will just add stress and urgency.

Remember that we talked about the speed of deployment and change. Deploying small increments often is the best way to ensure that each deployment will be of high quality. The constant flow of incremental work is very important.

Interrupting this flow due to an error is bad, but the effect multiplies quickly if the inability to deploy affects the pace of multiple microservices.

Multiple services being deployed simultaneously may also create a deadlock, where both services require work to fix the situation. This complicates the development and time to fix the issue.

Instead of simultaneous deployments, analysis needs to be done to determine which microservice is dependent on the other. Most of the time, it is obvious. In our example, the Frontend is dependent on the Thoughts Backend, so any change that involves them both will need to start with the Thoughts Backend and then move to the Frontend.

Actually, the Users Backend is a dependency of both, so assuming there's a change that affects the three of them, you'll need to first change the Users Backend, then the Thoughts Backend, and finally the Frontend.

Keep in mind that sometimes, it is possible that deployments need to move across services more than once. For example, let's imagine that we have a change in the signing mechanism for the authentication headers. The process then should be as follows:

1. Implement the new authentication system in the Users Backend, but keep producing tokens with the old system through a config change. The old authentication process is still used in the cluster so far.
2. Change the Thoughts Backend to allow working with both the old and the new system of authenticating. Note that it is not activated yet.
3. Change the Frontend to work with both authentication systems. Still, at this point, the new system is not yet used.
4. Change configuration in the Users Backend to produce new authentication tokens. Now is when the new system starts to be used. While the deployment is underway, some old system tokens may be generated.
5. The Users Backend and Frontend will work with any token in the system, either new or old. Old tokens will disappear over time, as they expire. New tokens are the only ones being created.
6. As an optional stage, the old authentication system can be deleted from the systems. The three systems can delete them without any dependency as the system is not used at this point.

At any step of the process, the service is not interrupted. Each individual change is safe. The process is slowly making the whole system evolve, but each of the individual steps is reversible if there's a problem, and the service is not interrupted.

Systems tend to develop by adding new features, and it is uncommon to have a clean-up stage. Normally, systems live with deprecated features for a long time, even after the feature is not used anywhere.

We will talk a bit more about clean-up in `Chapter 12`, *Collaborating and Communicating across Teams*.

This process may also be required for configuration changes. In the example, changing the private key required to sign the authentication headers will require the following steps:

1. Make the Thoughts Backend and Frontend able to handle more than one public key. This is a prerequisite and a new feature.
2. Change the handled keys in the Thoughts Backend to have both the old and the new public keys. So far, no headers signed with the new key are flowing in the system.
3. Change the handled keys in the Frontend to have both the old and the new. Still, no headers signed with the new key are flowing in the system.
4. Change the configuration of the Users Backend to use the new private key. From now on, there are headers signed with the new private key in the system. Other microservices are able to handle them.
5. The system still accepts headers signed with the old key. Wait for a safe period to ensure all old headers are expired.
6. Remove the configuration for the old key in the Users Backend.

Steps 2 to 6 can be repeated every few months to use new keys.

This process is called **key rotation**, and it is considered a good security practice as it reduces the life when a key is valid, reducing the window of time the system is vulnerable to a leaked key. For simplicity, we have not implemented it in our example system, but doing so is left as a recommended exercise. Try to change the example code to implement this key rotation example!

The full system feature may involve multiple services and teams. To help with coordinating the dependencies of the system, we need to know when a certain dependency of service is deployed and ready to go. We will talk about the inter-team communication in `Chapter 12`, *Collaborating and Communicating across Teams*, but we can help programmatically by making the service API to explicitly describe which version of the service is deployed, as we will discuss in the *Dealing with service dependencies* section.

In case there's a problem in the new version that has just been deployed, the deployment can be reverted quickly through a rollback.

Rolling back the microservices

Rollback is the process to step back quickly one of the microservices to the previous version.

This process can be triggered when there's a catastrophic error in a new version just released, so it can be solved quickly. Given that the version was already currently compatible, this can be done with confidence in a very short reaction time. Through GitOps principles, a `revert` commit can be done to bring back the old version.

The `git revert` command allows you to create a commit that undoes another, applying the same changes in reverse.

This is a quick way to undo a particular change, and to allow later to *revert the revert* and reintroduce the changes. You can check the Git documentation for more details (`https://git-scm.com/docs/git-revert`).

Given the strategic approach to keep moving forward, a rollback is a temporary measure that, while in place, will stop new deployments in the microservice. A new version addressing the bug that caused the catastrophic deployment should be created as soon as possible, so as to keep the normal flow of releases.

As you deploy more and more often, and get better checks in place, rollbacks will be less and less common.

Dealing with service dependencies

To allow services to check whether their dependencies have the correct version, we will make services to expose their version through a RESTful endpoint.

We will follow the example in the Thoughts Backend available in GitHub at this URL: `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter11/microservices/thoughts_backend`.

Check the version is available in the Frontend (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter11/microservices/frontend`).

The first step in the process is to properly define the version for each service.

Versioning the services

To allow a clear understanding of progress in our software, we need to name the different versions to be deployed. As we use `git` to keep track of changes, every commit in the system has an individual commit ID, but it doesn't follow any specific pattern.

To give a meaning to it and order them, we need to develop a version schema. There are multiple ways of making a version schema, including by release date (Ubuntu uses this one) or by `major.minor.patch`.

Having the same versioning scheme everywhere helps to develop a common language and understanding across teams. It also helps management to understand the changes—both in terms of when things are released, as well as how quickly they are changing. Agree with your teams a versioning scheme that makes sense in your organization, and follow it in all services.

For this example, we will use a `vMajor.Minor` schema and the version for the Users Backend as `v2.3`.

The most common pattern in software versioning is semantic versioning. This versioning pattern is useful for packages and customer-facing APIs, but less useful for internal microservice APIs. Let's see what its characteristics are.

Semantic versioning

Semantic versioning imposes meaning to the change on each of the different version numbers. This makes it easy to understand the scope of changes between versions and whether the update is risky to do on a dependent system.

Semantic versioning defines each version with three numbers: major, minor, and patch, normally described as `major.minor.patch`.

Increasing any of these numbers carries a particular meaning, as follows:

- Increasing the major number produces backward-incompatible changes.
- Increasing the minor number adds new features, but keeps backward-compatibility.
- Increasing the patch number fixes bugs, but doesn't add any new features.

As an example, Python works under this schema as follows:

- Python 3 included compatibility changes with Python 2.
- Python version 3.7 introduced new features compared with Python 3.6.
- And Python 3.7.4 added security and bug fixes compared with Python 3.7.3.

This versioning scheme is useful in communicating with external partners and is great for big releases and standard packages. But for small incremental changes in microservices, it is not very useful.

As we discussed in previous chapters, the key to continuous integration to be delivered is to make very small changes. They should not break backward compatibility, but, over time, old features will be dropped. Each microservice works in unison with other services, in a controlled manner. There's no need to have such a strong feature labeling, compared with an external package. The consumers of the service are the other microservices, under tight control in the cluster.

Some projects are abandoning semantic versioning due to this change in operation. For example, the Linux kernel stopped using semantic versioning to produce new versions without any particular meaning (`http://lkml.iu.edu/hypermail/linux/kernel/1804.1/06654.html`), as changes from one version to the next are relatively small.

Python will also treat version 4.0 as *the version that goes after 3.9*, without major changes like Python 3 had (`http://www.curiousefficiency.org/posts/2014/08/python-4000.html`).

That's why, internally, semantic versioning is *not* recommended. It may be useful to keep a similar versioning scheme, but without forcing it to make compatibility changes, just ever-increasing numbers, without specific requirements on when to change minor or major versions.

Externally, though, version numbers may still have a marketing meaning. For externally accessible endpoints, it may be interesting to use semantic versioning.

Once decided which version the service is, we can work on an endpoint that exposes this information.

Adding a version endpoint

The version to be deployed can be read from either the Kubernetes deployment or from the GitOps configuration. But there is a problem. Some of the configurations could be misleading or not uniquely pointing to a single image. For example, the `latest` tag may represent different containers at different times, as it gets overwritten.

Also, there's a problem of having access to the Kubernetes configuration or GitOps repo. For a developer, maybe this configuration is available, but they won't be for the microservices (nor should they be).

To allow the remainder of the microservices in the cluster to discover the version of the service, the best way is to explicitly create a version endpoint in the RESTful API. The discovery of the service version is granted, as it uses the same interface it will use in any other request. Let's see how to implement it.

Obtaining the version

To serve the version, we first need to record it into the service.

As we discussed previously, the version is stored as a Git tag. This will be our canon in the version. We will add the Git SHA-1 of the commit as well to avoid any discrepancies.

The SHA-1 is a unique ID that identifies each commit. It's produced by hashing the Git tree, so that it's able to capture any change—either the content or the tree history. We will use the full SHA-1 of 40 characters, even though sometimes it is abbreviated to eight or less.

The commit SHA-1 can be obtained with the following command:

```
$ git log --format=format:%H -n 1
```

This prints the last commit information, and only the SHA with the `%H` descriptor.

To get the tag this commit refers to, we will use the `git-describe` command:

```
$ git describe --tags
```

Basically, `git-describe` finds the closest tags to the current commit. If this commit is marked by a tag, as it should be for our deployments, it returns the tag itself. If it's not, it suffixes the tag with extra information about the commits until it reaches the current one. The following code shows how to use `git describe`, depending on the committed version of the code. Note how the code not associated with a tag returns the closest tag and extra digits:

```
$ # in master branch, 17 commits from the tag v2.3
$ git describe
v2.3-17-g2257f9c
$ # go to the tag
$ git checkout v2.3
$ git describe
v2.3
```

This always returns a version and allows us to check at a glance whether the code in the current commit is tagged in `git` or not.

Anything that gets deployed to an environment should be tagged. Local development is a different matter, as it consists of code that is not ready yet.

We can store these two values programmatically, allowing us to do it automatically and including them in the Docker image.

Storing the version in the image

We want to have the version available inside the image. Because the image is immutable, doing so during the build process is the objective. The limitation we need to overcome here is that the Dockerfile process does not allow us to execute commands on the host, only inside the container. We need to inject those values in the Docker image while building.

A possible alternative is to install Git inside the container, copy the whole Git tree, and obtain the values. This is usually discouraged because installing Git and the full source tree adds a lot of space to the container, something that is worse. During the build process, we already have Git available, so we just need to be sure to inject it externally, which is easy to do with a build script.

The easiest way of passing the value is through the `ARG` parameters. As a part of the build process, we will transform them into environment variables, so they'll be as easily available as any other part of the configuration. Let's take a look at the Dockerfile in the following code:

```
# Prepare the version
ARG VERSION_SHA="BAD VERSION"
ARG VERSION_NAME="BAD VERSION"
ENV VERSION_SHA $VERSION_SHA
ENV VERSION_NAME $VERSION_NAME
```

We accept an `ARG` parameter and then transform it into an environment variable through the `ENV` parameter. Both have the same name for simplicity. The `ARG` parameter has a default value for corner cases.

This makes the version available (inside the container) after we build it with the `build.sh` script, which obtains the values and calls `docker-compose` to build with the version as arguments, using the following steps:

```
# Obtain the SHA and VERSION
VERSION_SHA=`git log --format=format:%H -n 1`
VERSION_NAME=`git describe --tags`
# Build using docker-compose with arguments
docker-compose build --build-arg VERSION_NAME=${VERSION_NAME} --build-arg
VERSION_SHA=${VERSION_SHA}
# Tag the resulting image with the version
docker tag thoughts_server:latest throughs_server:${VERSION_NAME}
```

After the build process, the version is available as standard environment variables inside the container.

We included a script (`build-test.sh`) in each of the microservices in this chapter (for example, `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter11/microservices/thoughts_backend/build-test.sh`). This mocks the SHA-1 and version name to create a synthetic version for tests. It sets up the `v2.3` version for the Users Backend and `v1.5` for the Thoughts Backend. These will be used for examples in our code.

Check that the Kubernetes deployments include those versions (for example, the `https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/blob/master/Chapter11/microservices/thoughts_backend/docker-compose.yaml#L21` image is the `v1.5` version).

Furthermore, `VERSION_NAME` can also come from the CI pipeline as a parameter to the script. To do so, you'll need to replace the script to accept it externally, as seen in the `build-ci.sh` script:

```
#!/bin/bash
if [ -z "$1" ]
  then
    # Error, not version name
    echo "No VERSION_NAME supplied"
    exit -1
fi

VERSION_SHA=`git log --format=format:%H -n 1`
VERSION_NAME=$1

docker-compose build --build-arg VERSION_NAME=${VERSION_NAME} --build-arg
VERSION_SHA=${VERSION_SHA}
docker tag thoughts_server:latest throughs_server:${VERSION_NAME}
```

All the versions of these scripts include the tagging of the image with `VERSION_NAME` as a tag.

We can retrieve the environment variables with the version inside the container in the Python code, returning them in an endpoint, making the version easily accessible through the external API.

Implementing the version endpoint

In the `admin_namespace.py` file, we will create a new `Version` endpoint using the following code:

```
import os

@admin_namespace.route('/version/')
class Version(Resource):

    @admin_namespace.doc('get_version')
    def get(self):
        '''
        Return the version of the application
        '''
        data = {
            'commit': os.environ['VERSION_SHA'],
            'version': os.environ['VERSION_NAME'],
        }

        return data
```

Okay, now this code is very simple. It uses `os.environ` to retrieve the environment variables injected during the build as configuration parameters and return a dictionary with the commit SHA-1 and tag (described as a version).

The service can be built and run locally, using `docker-compose`. To test access to the endpoint in `/admin/version` and to check it, follow these steps:

```
$ cd Chapter11/microservices/thoughts_backend
$ ./build.sh
...
Successfully tagged thoughts_server:latest
$ docker-compose up -d server
Creating network "thoughts_backend_default" with the default driver
Creating thoughts_backend_db_1 ... done
Creating thoughts_backend_server_1 ... done
$ curl http://localhost:8000/admin/version/
{"commit": "2257f9c5a5a3d877f5f22e5416c27e486f507946", "version": "tag-17-
g2257f9c"}
```

As the version is available, we can update the autogenerated documentation to display the correct value, as shown in `app.py`:

```
import os
...
VERSION = os.environ['VERSION_NAME']
...

def create_app(script=False):
    ...
    api = Api(application, version=VERSION,
              title='Thoughts Backend API',
              description='A Simple CRUD API')
```

So the version gets properly displayed in the automatic Swagger documentation. Once the version for a microservice is accessible through an endpoint in the API, other external services can access it to discover the version and make use of it.

Checking the version

Being able to check the version through the API allows us to access the version easily in a programmatic way. This can be used for multiple purposes, like generating a dashboard displaying the different versions deployed in different environments. But we will explore the possibility to introduce service dependencies.

A microservice, when it starts, can check the services it depends on, and also checks whether they are higher than an expected version. If they're not, it will not start. This avoids configuration problems when one dependent service gets deployed before the dependency is updated. This can happen in complex systems where there's no great coordination in deployments.

To check the version, when starting the server in `start_server.sh`, we will first call a small script that checks the dependency. If it is not available, it will produce an error and stop. We will check that the Frontend has an available version of the Thought Backend or even higher.

The script, which we will call in our example, is called `check_dependencies_services.py`, and it gets called in `start_server.sh` for the Frontend.

The `check_dependencies_services` script can be divided into three parts: a list of the dependencies required; a check for one dependency; and a main section where each of the dependencies is checked. Let's take a look at the three parts.

Required version

The first section describes each of the dependencies and the minimum version required. In our example, we stipulate that `thoughts_backend` needs to be version `v1.6` or up:

```
import os

VERSIONS = {
    'thoughts_backend':
        (f'{os.environ["THOUGHTS_BACKEND_URL"]}/admin/version',
         'v1.6'),
}
```

This reuses the environment variable, `THOUGHTS_BACKEND_URL`, and completes the URL with the specific version path.

The main section goes through all the dependencies described to check them.

The main function

The main function iterates through the `VERSIONS` dictionary and, for each one, does the following:

- Calls the endpoint
- Parses the result and gets the version
- Calls `check_version` to see whether it's correct

If it fails, it ends with a `-1` status, so the script reports as failed. These steps are executed through the following code:

```
import requests

def main():
    for service, (url, min_version) in VERSIONS.items():
        print(f'Checking minimum version for {service}')
        resp = requests.get(url)
        if resp.status_code != 200:
            print(f'Error connecting to {url}: {resp}')
            exit(-1)

        result = resp.json()
        version = result['version']
        print(f'Minimum {min_version}, found {version}')
        if not check_version(min_version, version):
            msg = (f'Version {version} is '
```

```
                    'incorrect (min {min_version})')
            print(msg)
            exit(-1)

if __name__ == '__main__':
    main()
```

The main function also prints some messages to help to understand the different stages. To call the version endpoint, it uses the `requests` package and expects both a `200` status code and a parsable JSON result.

Note that this code iterates through the `VERSION` dictionary. So far, we only added one dependency, but the User Backend is another dependency and can be added. It's left as an exercise to do.

The version field will be checked in the `check_version` function, which we will see in the next section.

Checking the version

The `check_version` function checks whether the current version returned is higher or equal to the minimum version. To simplify it, we will use the `natsort` package to sort the versions and then check the lowest.

You can check out the `natsort` full documentation (`https://github.com/SethMMorton/natsort`). It can sort a lot of natural strings and can be used in a lot of situations.

Basically, `natsort` supports ordering common patterns of versioning, which includes our standard versioning schema described previously (`v1.6` is higher than `v1.5`). The following code uses the library to sort both versions and verify that the minimum version is the lower one:

```
from natsort import natsorted

def check_version(min_version, version):
    versions = natsorted([min_version, version])
    # Return the lower is the minimum version
    return versions[0] == min_version
```

With this script, we can now start the service and it will check whether the Thoughts Backend has the proper version. If you started the service as described in the *Technical requirements* section, you'll see that the Frontend is not starting properly, and produces a `CrashLoopBackOff` status, as shown here:

```
$ kubectl get pods -n example
NAME READY STATUS RESTARTS AGE
frontend-54fdfd565b-gcgtt 0/1 CrashLoopBackOff 1 12s
frontend-7489cccfcc-v2cz7 0/1 CrashLoopBackOff 3 72s
grafana-546f55d48c-wgwt5 1/1 Running 2 80s
prometheus-6dd4d5c74f-g9d47 1/1 Running 2 81s
syslog-76fcd6bdcc-zrx65 2/2 Running 4 80s
thoughts-backend-6dc47f5cd8-2xxdp 2/2 Running 0 80s
users-backend-7c64564765-dkfww 2/2 Running 0 81s
```

Check the logs of one of the Frontend pods to see the reason, using the `kubectl logs` command, as seen here:

```
$ kubectl logs frontend-54fdfd565b-kzn99 -n example
Checking minimum version for thoughts_backend
Minimum v1.6, found v1.5
Version v1.5 is incorrect (min v1.6)
```

To fix the problem, you need to either build a version of the Thoughts Backend with a higher version or reduce the dependency requirement. This is left as an assessment at the end of the chapter.

Summary

In this chapter, we learned how to deal with elements that work with several microservices at the same time.

First, we talked about strategies to follow when new features need to change multiple microservices, including how to deploy small increments in an ordered fashion and to be able to roll back if there's a catastrophic problem.

We then talked about defining a clear versioning schema, and adding a version endpoint to the RESTful interfaces that allow self-discovery of the version for microservices. This self-discovery can be used to ensure that a microservice that depends on another is not deployed if the dependency is not there, which helps in coordinating releases.

The code in GitHub for the Frontend in this chapter (`https://github.com/PacktPublishing/Hands-On-Docker-for-Microservices-with-Python/tree/master/Chapter11/microservices/frontend`) includes a dependency to the Thoughts Backend that will stop deploying it. Note that the code, as is, won't work. Fixing it is left as an exercise.

We also learned how to use ConfigMap to describe configuration information that's shared across different services in the Kubernetes cluster. We later described how to use Kubernetes secrets to handle a configuration that's sensitive and requires extra care.

In the next chapter, we will see the various techniques for coordinating, in a highly effective manner, different teams working with different microservices.

Questions

1. What are the differences between releasing changes in a microservice architecture system and a monolith?
2. Why should the released changes be small in a microservice architecture?
3. How does semantic versioning work?
4. What are the problems associated with semantic versioning for internal interfaces in a microservice architecture system?
5. What are the advantages of adding a version endpoint?
6. How can we fix the dependency problem in this chapter's code?
7. Which configuration variables should we store in a shared ConfigMap?
8. Can you describe the advantages and disadvantages of getting all the configuration variables in a single shared ConfigMap?
9. What's the difference between a Kubernetes ConfigMap and a Kubernetes secret?
10. How can we change a Kubernetes secret?
11. Imagine that, based on the configuration, we decided to change the `public_key.pub` file from a secret to a ConfigMap. What changes do we have to implement?

Further reading

For handling your secrets on AWS, you can interact with a tool called CredStash (`https://github.com/fugue/credstash`). You can learn more about how to use it in the book *AWS SysOps Cookbook – Second Edition* (`https://www.packtpub.com/cloud-networking/aws-administration-cookbook-second-edition`).

12
Collaborating and Communicating across Teams

As we discussed previously, the main characteristic of microservices is the ability to develop them in parallel. To ensure maximum efficiency, we need to coordinate our teams successfully to avoid clashes. In this chapter, we will talk about the different elements we need to understand to ensure that the different teams work together successfully.

First, we will cover how to get a consistent vision across different microservices, how the different communication structures shape communication in the software elements, and how to ensure that we don't accumulate cruft in the software. Then, we will talk about how to ensure that teams coordinate themselves on releases and refine their processes and tools to make them more and more reliable.

The following topics will be covered in this chapter:

- Keeping a consistent architectural vision
- Dividing the workload and Conway's Law
- Balancing new features and maintenance
- Designing a broader release process

By the end of this chapter, we will know how to structure and coordinate different teams that work independently so that we can get the most out of them.

Keeping a consistent architectural vision

In a system structured on microservices, each team is able to perform most of the tasks on their own, independently from other teams. Designing the services so that they are as independent as possible and with minimal dependencies is key to achieving a good development speed.

Therefore, microservice separation allows teams to work independently and in parallel, while with monoliths, most of the people working on it keep track of what goes on, even to the point of being distracted with work out of the field of focus of a particular developer. They'll know when a new version is released and see new code being added to the same code base they are working on. However, that's not the case in the microservices architecture. Here, teams focus on their services and are not distracted by other features. This brings clarity and productivity.

However, a global vision of the system is still required. There's a need for a long-term view on how the architecture of the system should change over time so that it can be adapted. This vision (in monolithic systems) is implicit. Microservices need to have a better understanding of these changes so that they can work effectively, so a leading architect who can unify this global vision is very important.

The architect's role is a position in the software industry that isn't defined consistently.

In this book, we will define it as a role that deals with the structure of APIs and services as a whole. Their main objective is to coordinate teams when it comes to technical issues, rather than dealing with code directly.

Explicitly naming someone who's responsible for the global architecture of the system helps us keep a long-term vision of how the system should evolve.

In small companies, Chief Technical Officers may fulfill the architect's role, though they will also be busy handling elements that are related to managerial processes and costs.

The main responsibility of a leading architect is to ensure that the microservices division keeps making sense as it evolves and that the APIs that communicate between services are consistent. They should also try to facilitate the generation of standards across teams and share knowledge across the organization.

The architect should also be the final decision maker when it comes to any questions regarding what feature goes with what microservice, as well as any other conflicts that may arise that involve several teams. This role is of great help during the transition from a monolith to a microservice architecture, but after that process is completed, they can also ensure that the organization can adapt to new challenges and keep technical debt under control. A system working in a microservices architecture aims to create independent teams, but they all truly benefit from a shared global vision that's created by an external person.

To allow for better coordination, how the teams are divided is highly important. Let's learn about some of the challenges that arise when we divide the development of a system into different teams.

Dividing the workload and Conway's Law

Microservice architecture systems are adequate for big software systems, though companies tend to start with monolithic applications. This makes sense for any system that has small teams. As the system is explored and pivoted, it grows over time.

But when monolith systems grow to a certain size, they become difficult to handle and develop. The internals become intertwined for historical reasons, and with increased complexity, the reliability of the system can become compromised. Finding the balance between flexibility and redundancy can be difficult.

Remember that microservices are useful when the development team is big. For small teams, a monolith is easier to develop and maintain. It's only when many developers work on the same system that dividing the work and accepting the overheads of a microservice architecture makes sense.

Scaling the development team can become difficult as there will be too much old code there, and learning how to navigate through it is difficult and takes a lot of time. The developers (the ones who have been around for a long time) know what caveats can help, but they become bottlenecks. Increasing the size of the team doesn't help because making any change can become complicated. Therefore, every new developer needs a lot of training before they can get up to speed and be able to successfully work on bug fixes and new features.

Teams also have a natural size limit. Growing beyond that limit can mean having to split them into smaller ones.

> The size of a team is highly variable, but normally, the 7±2 components are considered as a rule of thumb for the ideal number of people who should be in a team.

Bigger groups tend to generate smaller groups on their own, but this means there will be too many to manage and some may not have a clear focus. It's difficult to know what the rest of the team is doing.

Smaller teams tend to create overhead in terms of management and inter-team communication. They'll develop faster with more members.

In a big monolith system, multiple independent teams will tend to mess around without a clear long-term view. This may be alleviated by designing a robust internal structure, but it will require huge up-front planning and strong policing to ensure it is being followed.

The microservice architecture is a design that tackles these problems as it makes very strict boundaries between parts of the system. However, doing so requires the development team to be of a certain size so that they can work as several small teams working mostly independently. This is the main characteristic of a microservice architecture system. Each of the microservices that form it is an independent service that can be developed and released independently.

This way of working allows teams to work in parallel, without any interference. Their realm of action is clear, and any dependency is explicitly set. Due to this, the borders between the microservices are strong.

Just because a microservice is released independently doesn't imply that a single release is enough to release a full feature. As we've already seen, sometimes, a feature in a microservice requires that another microservice is worked on before it can be deployed. In this case, several microservices need to be worked on.

The most important idea to keep in mind when planning on how to divide teams is how the team's structure is reflected in software. This is described by Conway's Law.

Describing Conway's Law

Conway's Law is a software adage (`https://www.nagarro.com/en/blog/post/76/microservices-revisiting-conway-s-law`). In other words, in any organization that produces software, the software will replicate the communication structure of the organization. For example, in a very simplified way, an organization is divided in two departments: purchases and sales. This will generate two software blocks: one that's focused on buying and another that's focused on selling. They will communicate when required.

In this section, we will talk about *software units*. This is a generic term that describes any software that's treated as a single cohesive element. It can be a module, a package, or a microservice.

In the microservice architecture, these software units are mainly microservices, but in some cases, there can be other types. We will see examples of this in the *Dividing the software into different kinds of software units* section.

This may be unsurprising. It's only natural that communication levels are different between teams, as well as within the same team. However, the implications of teamwork are enormous, some of which are as follows:

- Inter-team APIs are more expensive than intra-team APIs, both in terms of operating them as well as developing them since their communication is more complicated. It makes sense to make them generic and flexible so that they can be reused.
- If the communication structures replicate the human organization, it makes sense to be explicit. Inter-team APIs should be more visible, public, and documented than intra-team APIs.
- When designing systems, dividing them across the lines of the layered team structure is the path of least resistance. Engineering them any other way would require organizational changes.
- On the other hand, changing the structure of an organization is a difficult and painful process. Anyone that has gone through a reorg knows this. The change will be reflected in the software, so plan accordingly.

- Having two teams working on the same software unit will create problems because each one will try to pull it to their own goals.

The owner of a software unit should be a single team. This shows everyone who's responsible for who has the final say on any change and helps us focus on our vision and reduce technical debt.

- Different physical locations impose communications restrictions, such as time differences, which will produce barriers when we're developing software across them. It is common to split teams by location, which creates the need to structure the communication (and therefore the APIs) between these teams.

Note that the DevOps movement is related to Conway's Law. The traditional way of dividing the work was to separate the software being developed from how it's run. This created a gap between both teams, as described by Conway's Law, which generates problems related to a lack of understanding between the two teams.

The reaction to this problem was to create teams that could develop and operate their own software, as well as deploy it. This is called DevOps. It shifted the operational problems to the development team with the aim of creating a feedback loop to incentivize, understand, and fix them.

Conway's Law is not a bad thing to overcome. It's a reflection that any organizational structure has an impact on the structure of software.

Remembering this may help us design the system so that the communication flow makes sense for the organization and existing software.

One of the key components of the DevOps movement was advancing the technology that built systems in order to simplify how production environments operated so that the deployment process was simpler. This allows us to structure teams in new ways, thereby leading to multiple teams being able to take control of releases.

Now, let's talk about how the software can be structured into different divisions.

Dividing the software into different kinds of software units

While the main objective of this book is to talk about the division of software in microservices, this isn't the only possible division. Other divisions can include modules inside a microservice or a shared package.

The main characteristic of a microservice is that it is independent in terms of development and deployment, so full parallelization can be achieved. Other divisions may reduce this and introduce dependencies.

Ensure that you justify these changes.

In the example system that we introduced in this book, we introduced a module that verifies that a request is signed by a user. The Users Backend generates a signed header and the Thoughts Backend and Frontend verify it independently through the `token_validation.py` module.

This module should be owned by the same team that owns the Users Backend since it's a natural extension of it. We need to verify that it generates the same tokens that the Users Backend generates.

The best way to avoid duplication and to keep it always in sync is to generate a Python package that can be installed on the dependent microservices. Then, the packages can be treated like any other external dependency in the `requirements.txt` file.

To package a library in Python, we can use several tools, including those in the official *Python Packaging User Guide* (`https://packaging.python.org/`) to newer tools such as Poetry (`https://poetry.eustace.io`), which are easier to use for a new project.

The package can be uploaded to PyPI if we want it to be publicly available. Alternatively, we can upload it to a private repository either using a tool such as Gemfury or by hosting our own repository if required. This makes an explicit division between the package and its maintainers, as well as the teams that are using it as a dependency.

Dividing the software units has implications in terms of team division. Now, let's take a look at how to structure teams.

Designing working structures

Taking Conway's Law into account, dividing the software should reflect how the organization is structured. This is very relevant when we're migrating from a monolith to a microservice architecture.

Remember, moving from a monolith to a microservice is a big change in terms of how we operate. It is as much an organizational change as it is a technology change. The main risks are in the human component, including challenges such as training people to use the new technologies and keeping the developers happy with the new area they'll be working in.

Making a radical change to the organization's structure can be very difficult, but small tweaks will be required. When making a big change when migrating from a monolith to a microservice, the teams will need to be restructured.

Keep in mind that a big reorg has the potential of getting people angry and causing political problems. Humans don't like change, and any decision needs to make sense. Expect to have to explain and clarify this move. Having clear objectives regarding what to achieve with the new structure will help give it purpose.

Let's take a look at some examples of team divisions, as well as their pros and cons.

Structuring teams around technologies

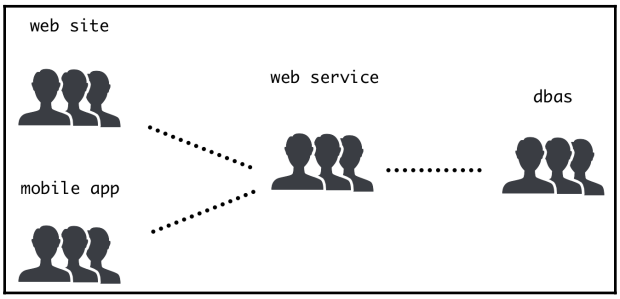
In some cases, different skills related to technology may be relevant. Some parts of the system may deal with a technology that's totally different from anything else.

A very good example is mobile applications since they are restrictive in terms of the languages they use (Java for Android and Objective-C or Swift for iOS). An application with a website and a mobile app may require a specific team to work on the mobile app's code.

A more traditional example is the database team, which was built around **Database Administrators** (**DBAs**). They would control access to databases and operate them to keep them in good shape. This structure is disappearing, though, since database operations are now easier and typically handled by most developers, and the infrastructure management for databases has been greatly simplified in recent years.

This may allow us to justify creating specific teams around certain areas. The technology's barriers ensure that the communication between systems is structured.

The following diagram is an example of the kinds of teams we'll come across. They're grouped by technology and how they'll communicate. The database team will communicate with the team that's creating the web service backend, and they will communicate with the web and mobile teams:



The main disadvantage of this model is that new features are likely to require multiple teams to work on them. Any change that's made to customer-facing code so that we can store a new value in the database requires work input from every team. These features require extra coordination, which can limit the speed of development.

Structuring teams around domains

Another structure is the one that surrounds different knowledge domains and is typically related to business areas in the company. Each knowledge domain has its own self-contained system, but they communicate with each other. Some parts may have an externally accessible interface, while others may not.

This kind of structure is typically found in established organizations that have different areas that they have been working successfully for years.

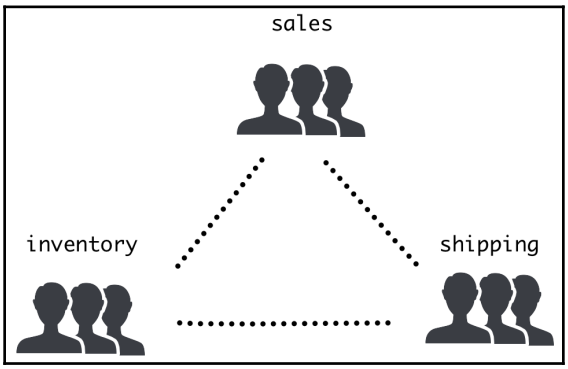
For example, an online retailer may be structured in three areas:

- **Sales**: Handles the external website and marketing.
- **Inventory**: Purchases the merchandise so that it can be sold, and also handles stock.
- **Shipping**: Delivers the product to the customers. The tracking information is displayed on the website.

In this case, each area has its own database so that it can store the relevant data and its service. They communicate with each other with defined APIs, and the most frequent changes happen within a domain. This allows for quick releases and development within the domains.

Having new features across domains is also possible. For example, a change in the tracking information for shipping may require us to match the changes that are produced by sales. However, these changes should happen less often.

In this example, each of the teams will communicate with each other, as shown in the following diagram:



The main inconvenience of this structure is the possibility of creating isolated teams and a silo mentality. Each system has its own way of doing things, so they can diverge to the point of not sharing the same basic language. When a cross-domain feature is required, it can lead to discussion and friction.

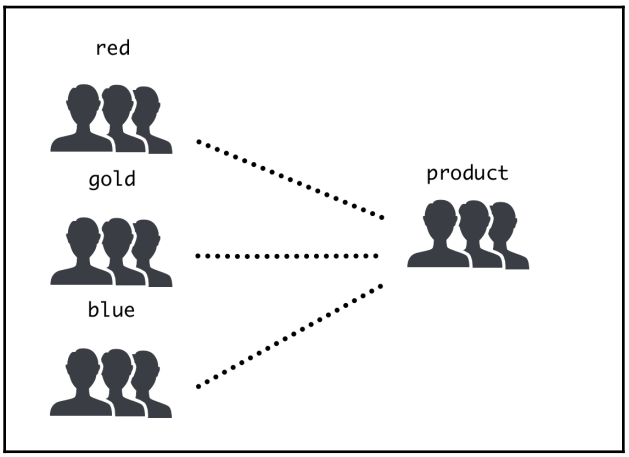
Structuring teams around customers

In some organizations, the main objective is to create custom work for customers. Maybe the customers need to integrate with the product in custom B2B ways. Being able to develop and run custom code is critical in this scenario.

The structure is focused on customers. Three teams (called red, gold, and blue) are assigned to customers and maintain a special service for each of them, including their custom code. Each customer team handles a couple of customers. Another team handles the product's backend, which contains the common code and infrastructure for the system. This team works separately from the customers but adds features from the customer teams when they are shared so that they're included in the product. General improvements are also shared.

This creates two speeds in the organization. The customer teams are focused on the short-term demands of customers, while the product team is focused on the long-term demands of customers.

Here, the product team will talk to the customer teams, but the customer teams won't talk to each other that much. This is shown in the following diagram:



This structure works well for highly customized services so that they can contain the code that's been generated for a single customer that could make them lose focus on the general product. The main problem here is the high pressure that the customer teams may face as they are exposed to demanding customers, and that can be taxing for developers. The product team needs to ensure that they are making useful additions to the product and reducing their long-term problems as much as possible.

Structuring teams around a mix

The previous three examples are synthetic use cases. Real life is more complicated and may require a mix of all of them, or a new structure altogether.

If the organization is big enough, there may be dozens of different teams and software units. Remember that a single team can handle more than one software unit if it is big enough. However, two teams shouldn't own the same software unit in order to avoid ownership and a lack of focus.

Analyze the flow of communication in your organization in order to understand the pain points to address when you're moving to microservices and be sure that the human structure is taking designing the microservices and software units into account.

Another important element for teams is to find a proper balance between time that's spent on adding new features and time maintaining the existing code.

Balancing new features and maintenance

Every software service requires maintenance so that it stays in good shape, but don't add evident external value. Maintenance tasks are critical for good operation, though, and can be divided into two categories: regular maintenance and managing technical debt.

Technical debt is the one that will use most of the time and requires further discussion, but before that, let's take a look at regular maintenance.

Regular maintenance

This kind of maintenance comes in the shape of tasks that are inherent to the nature of a software service. By running a service that depends on other components, such as underlying operative systems or the Python interpreter, we need to keep them up to date and upgrade them to new versions.

In the context of using containers and Kubernetes, there are two systems that act as operating systems that we need to take into account. One is the OS from the containers; here, we used Alpine. The other is the OS that handles the Kubernetes nodes, in which AWS EKS is handled automatically, but needs to be upgraded to the Kubernetes version.

The main reasons to keep the dependencies up to date are as follows, in their order of importance:

- New versions fix security problems.
- General performance improvements.
- New features can be added that enable new functionality.

These kinds of tasks can be mitigated if we plan for them. For example, using a version of the operative system labeled **Long-Term Support** (**LTS**) can reduce problems when it comes to updating the system.

An LTS version of an OS is a version that receives support and critical updates during a long cycle. For example, a regular Ubuntu version is released every 6 months and receives updates (including critical security updates) for 9 months. LTS versions are released every 2 years and receive support for 5 years.

When running services, it is recommended to use LTS versions in order to minimize required maintenance work.

All of these packages and dependencies need to be updated to ensure that the OS works well. The alternative is to have open security holes or be left with an outdated system.

Updating dependencies could require us to adapt the code, depending on whether parts of it become deprecated or are removed. This can be costly in some cases. At the time of writing, the most famous migration has been the upgrade from Python 2 to Python 3 by the Python community, a task that is taking multiple years.

Most of the upgrades are normally pretty routine and require minimal work, though. Try to generate an upgrade plan that keeps pace in a sensible manner and produces solid guidelines; for example, rules such as *when a new operative system LTS version is released* and *all systems should be migrated in the following 3 months*. This produces predictability and gives everyone a clear objective that can be followed up and enforced.

Continuous integration tools can help in this process. For example, GitHub automatically detects dependencies in files such as `requirements.txt` and notifies us when a vulnerability is detected. It's even possible to automatically generate pull requests when updating modules. Check out the documentation for more information: `https://help.github.com/en/github/managing-security-vulnerabilities/configuring-automated-security-fixes`.

Upgrading dependencies is probably the most common regular maintenance task, but there are other possibilities:

- To clean up or archive old data. These operations can normally be automated, saving a lot of time and reducing problems.
- To fix operations that are dependent on business processes, such as generating monthly reports, and so on. These should be automated when possible, or tools should be designed so that users can produce them automatically instead of relying on technical staff doing bespoke operations.
- To fix permanent problems that are produced by bugs or other errors. Bugs sometimes leave the system in a bad state; for example, there may be a corrupted entry in the database. While the bug is being fixed, we may need to resolve the situation by unblocking a process or a user.

These processes can be annoying, especially if they are repetitive, but are normally well understood.

The other form of maintenance, which deals with technical debt, is more complicated, as it is introduced more gradually and is more difficult to detect clearly. Properly addressing technical debt is the most challenging maintenance task, but before we can do anything, we need to understand it.

Understanding technical debt

Technical debt is a concept that's used in software development to describe additional costs that will be added in the future when a non-optimal solution is implemented. In other words, choosing the quick or easy choice means later features take longer and are more difficult to develop.

As a metaphor, technical debt has been around since the early 90s, but the concept has been described before then.

Like any metaphor, it is useful, but it has limits. In particular, non-technical people tend to associate it with financial debt, even though they have different implications. For example, most technical debt is created without us even noticing it. Make sure that you don't take the metaphor too far.

Technical debt is unavoidable to a certain degree. There's no infinite time to research all the possibilities before implementing a feature and there's no perfect information at the time of making any decision. It is also a consequence of entropy growth in any complex system.

Apart from being unavoidable, it can also be a deliberate choice. Development is constrained by time, so an imperfect quick solution to market may be preferable to missing a deadline.

Another sign of technical debt is concentrating on certain knowledge. In any case, technical debt keeps piling up over time, and this creates friction for new features. Complexity increases can also create reliability problems as bugs will be more and more difficult to understand and fix.

Simplicity is the best friend of reliable systems. Simple code is easy to understand and correct and makes bugs either obvious or quick to detect. The microservice architecture aims to reduce the inherent complexity of a monolith by creating independent services that are smaller and have clear responsibilities assigned to them and that create explicit interfaces across them.

Technical debt can grow to a point where a big architecture is required. We've seen how moving from a monolith into a microservices architecture could be one of these moments.

An architectural migration such as this is a big effort and will require time to deliver. New microservices that are reproducing the features that already exist in the monolith may conflict with new features being introduced.

This creates a moving target effect that can be very disruptive. Ensure that you identify these conflicting points and try to minimize them in your migration plan. Some new features may be able to be delayed until the new microservice is ready, for example.

However, instead of waiting until the technical debt is so big that only radical changes are good enough to address it, we need to be able to tackle technical debt earlier.

Continuously addressing technical debt

Decreasing technical debt needs to be a continuous process and one that's introduced into the day-to-day operation of things. Agile techniques, which focus on continuous improvement, try to introduce this kind of mentality.

Detecting technical debt will typically come from within the development team since they are the ones closer to the code. Teams should be thinking about where operations could be smoother and have reserved time to perform those improvements.

A great source of information that allows us to detect technical debt is metrics, such as the ones we set up in `Chapter 10`, *Monitoring Logs and Metrics*.

The risk of neglecting to fix these issues is to fall into software rot when already existing features are slowly getting slower and less reliable. Over time, they'll be more and more obvious to customers and external partners. Way before that, working in this kind of environment will make a developer's life difficult, and there's a risk of burnout. Delays in new developments will also be common as code will be inherently difficult to work with.

To avoid getting into this situation, time needs to be allocated to reduce technical debt in a continuous fashion, intercalated with new features and other work. A balance should be found between maintenance and technical debt reduction and new features.

A lot of the techniques we've talked about in this book help us improve the system in a continuous fashion, from the continuous integration techniques we described in `Chapter 4`, *Creating a Pipeline and Workflow*, to the code reviews and approvals that we described in `Chapter 8`, *Using GitOps Principles*.

Distribution may be highly dependent on the current shape of the system, but it really helps that it is explicit and enforced. Something such as a specific time percentage to be spent on technical debt reduction could be helpful.

Reducing technical debt is costly and difficult, so introducing as little as possible makes sense.

Avoiding technical debt

The best way to deal with technical debt is to not introduce it in the first place. However, this is easier said than done. There are multiple elements that can affect the quality of the decisions that lead to technical debt.

The most common causes are as follows:

- **Lack of a strategic, high-level plan to give direction**: This produces inconsistent results because each time the same problem is found, it will be resolved in a different way. We talked about how coordination across teams needs to address standards across the organization and ensure they are followed. Having someone acting as a software architect, looking for creating consistent guidelines across the board, should greatly improve this case.
- **Not having the proper knowledge to choose the right option**: This is quite common. Sometimes, the people that need to make a decision don't have all the relevant pieces of information due to miscommunication or simply lack of experience. This problem is typical of structures that lack experience in the problems at hand. Ensuring that you have a highly trained team and are creating a culture where more experienced members help and mentor junior members will reduce these cases. Documentation that keeps track of previous decisions and simplifies how to use other microservices will help us coordinate teams so that they have all the relevant parts of the puzzle. This helps them avoid making mistakes due to incorrect assumptions. Another important element is to ensure that teams have proper training in the tools they're using so that they are fully aware of their capacities. This should be the case for external tools, such as being skilled in Python or SQL, and in any internal tool that requires training materials, documentation, and appointed points of contact.

- **Not spending enough time investigating different options or planning**: This problem is created by pressure and by the need to make quick progress. This can be ingrained in the organization culture, and slowing down decision-making could be a challenging task when the organization grows since smaller organizations tend to require a faster process. Documenting the decision process and requiring it to be peer-reviewed or approved can help slow this down and ensure that work is thorough. It's important to find a balance in terms of what decisions require more scrutiny and which ones don't. For example, everything that fits neatly inside a microservice can be reviewed inside the team, but features that require multiple microservices and teams should be reviewed and approved externally. In this scenario, finding the proper balance between gathering information and making a decision is important. Document the decisions and the inputs so that you understand the process that got them there and refine your process.

The best way to avoid these problems is to reflect on previous errors and learn from mistakes.

Designing a broader release process

While the ability to deploy each microservice independently is really a key element of the system, this doesn't mean that no coordination should be required.

First, there are still some features that need to be deployed in multiple microservices. We've already looked at how we can work on the development process, including details such as handle versioning and checking dependencies explicitly. So what now?

In these situations, coordination between teams is required to ensure that the dependencies are implemented and that the different deployments are executed in an adequate order.

While some coordination can be helped by the leading architect, the architecture role should be focused on long-term goals and not short-term releases. A good tool to allow self-coordination by teams is to inform the other teams in a meeting about releases.

Planning in the weekly release meeting

When the release process is new and migrating from the monolith is still underway, it is a good idea to provide insight into every team is doing. A weekly release meeting that is attended by representatives of every team can be an excellent way of distributing knowledge regarding what is going on in other teams.

The objectives for the release meeting should be as follows:

- Planned releases for the next 7 days and rough times for when; for example, we plan to release a new version of the Users Backend on Wednesday.
- You should provide a heads-up for any important new feature, especially if other teams can use it. For example, if the new version improves authentication, make sure that you redirect your teams to the new API so that they can get these improvements as well.
- State any blockers. For example, we can't release this version until the Thoughts Backend releases their version with feature A.
- Raise any flags if there's critical maintenance or any changes that could affect the releases. For example, on Thursday morning, we need to do database maintenance, so please don't release anything until 12 o'clock. We will send an email when the work is done.
- Review the release problems that happened the week prior. We'll talk about this in more detail later.

This is similar to the usual standup meetings that are present in many agile practices, such as SCRUM, but focused on releases. To be able to do this, we need to specify when the release is happening in advance.

Given the asynchronous nature of microservices releases, and as continuous integration practices are implemented and speed up this process, there will be a lot of routine releases that won't be planned with that much time in advance. This is fine and means that the release process is being refined.

Try to plan a bit in advance when it comes to riskier releases and use the release meeting to communicate with the rest of the teams effectively. The meeting is a tool that keeps the conversation open.

Over time, as continuous integration practices become more and more established and the releases become quicker and quicker, the weekly release meeting will slowly become less and less important, to the point that it may not need to be done anymore – at least not as regularly. This is part of reflecting on practices for continuous improvement, which is also achieved by identifying release problems.

Reflecting on release problems

Not every release will go perfectly fine. Some will fail due to problems in the tools or infrastructure, or maybe because there's a mistake in the process that's easy to make. The fact is that some releases will have issues. Unfortunately, it's not possible to avoid these situations.

To reduce and minimize release issues over time, each time a problem is found, it needs to be documented and brought up in the weekly release meeting or in an equivalent forum.

Some problems will be minor and require only a bit of extra work for the release to be successful; for example, a misconfiguration that avoids the new release from being started until it's fixed or a coordination problem where one service is deployed before its dependency.

Other problems will be greater, maybe even causing an outage due to a problem. Here, rollbacks will be useful so that we can quickly go back to a known state and plan again.

In any case, they should be properly documented, even if this is just briefly, and then shared so that the process can be refined. Analyzing what went wrong is key to keep improving your releases so that they're faster and less complicated.

Be open about these problems. Creating a culture where problems are openly debated and acknowledged is important if you wish to detect every single problem and quickly assess solutions.

Capturing problems is not, and should never be, about assigning blame. It's the organization's responsibility to detect and correct problems.

If this happens, not only will the environment become less attractive to work in, but problems will be hidden by teams so that they don't get blamed.

Unaddressed problems tend to be multiplicative, so reliability will suffer greatly.

Being able to release uneventfully is critical for quick deployments and to increase speed. Only light documentation is required when we're dealing with these kinds of problems since they are normally mild and, in the worst case, delay a release for a day or two.

For bigger problems, when the external service is disrupted, it's better to have a more formal process to ensure that the problem is fixed properly.

Another way we can improve the process is to properly understand the causes of problems that interrupt the service in live systems. The most effective tools for this are post-mortem meetings.

Running post-mortem meetings

Not limited to releases, sometimes, there will be big events that interrupt the service and need major work so that they can be fixed. In these emergency situations, the first objective is to recover the service as soon as possible.

After the service is stable again, to learn from this experience and to avoid it happening again, a post-mortem meeting should be attended by everyone that was involved in the incident. The objective of the post-mortem meeting is to create a series of follow-up tasks from the lessons that were learned during the emergency.

To document this, you need to create a template. This will be filled in during the post-mortem meeting. The template should capture the following information:

- **What problem was detected?** Include how it was detected if this isn't evident; for example, the website was down and was returning 500 errors. This shows that there was an increase in errors.
- **When did it start and finish?** A timeline of the incident; for example, Thursday from 3PM to 5PM.
- **Who was involved in remediating the incident?** Either detecting the problem or fixing it. This helps us collect information about what happened.
- **Why did it fail?** Go to the root cause and the chain of events leading to that; for example, the website was down because the application couldn't connect to the database. The database was unresponsive because the hard drive was full. The hard drive was full because the local logs filled up the disk.
- **How was it fixed?** Steps were taken to solve the incident; for example, logs older than a week were deleted.
- **What actions should follow up from this incident?** Actions that should remediate or fix the different issues. Ideally, they should include who will perform the action; for example, no local logs should be stored and they should be sent to the centralized log. The amount of hard disk space should be monitored and alert if less than 80% of the space is available.

Some of these elements can be filled in immediately after the emergency, such as who was involved. However, it's good to schedule the post-mortem meeting one to three days after it happened to allow everyone to digest and process this data. The root cause can be different to what we initially thought, and spending some time thinking about what happened helps us come up with better solutions.

As we discussed in the *Reflecting on release problems* section, be sure to encourage open and candid discussion when you're dealing with service interruption incidents.

Post-mortem meetings are not there to blame anyone, but to improve the service and reduce risks when you're working as a team.

The follow-up actions should be decided in the meeting and prioritized accordingly.

Although detecting the root cause is very important, note that actions should be taken against other causes. Even if the root cause is one, there are also other preventive actions that can minimize the effect it has if it happens again.

Actions that come about from post-mortem meetings are typically high priority and should be done as soon as possible.

Summary

In this chapter, we've looked at the different aspects of coordination across teams so that we can successfully manage an organization running a microservice architecture.

We started by talking about how keeping a global vision and coordination between parts is good. We talked about having an explicitly named leading architect that oversees the system and has a high-level view that allows them to ensure that teams aren't conflicting with each other.

We described Conway's Law and how the communication structure ends up shaping the software's structure so any change made to the software should be reflected somehow in the organization and vice versa. Then, we learned how to divide areas of responsibility and provided some examples of possible divisions, based on different organizations.

Next, we introduced how technical debt slows down the ongoing development process and how it's important to introduce a mindset of continuously addressing it to avoid degrading the experience for the internal teams working on them and the customers who are using them.

Finally, we addressed some of the problems that releases may cause, both in terms of coordinating adequately between teams, especially in the early stages of working with GitOps, and making retrospective analysis when releases fail or when services are down.

Questions

1. Why is a leading architect convenient for a microservice-architecture system?
2. What is Conway's Law?
3. Why does technical debt get introduced?
4. Why is it important to create a culture where we can work continuously to reduce technical debt?
5. Why is it important to document problems in releases and share them with every team?
6. What is the main objective of a post-mortem meeting?

Further reading

To learn more about the role of the architect, read *Software Architect's Handbook* (`https://www.packtpub.com/application-development/software-architects-handbook`), which includes chapters devoted to soft skills and architecture evolution. You can read more about Conway's Law and structuring digital businesses in *The New Engineering Game* (`https://www.packtpub.com/data/the-new-engineering-game`).

Assessments

Chapter 1

1. **What is a monolith?**

   A monolith is a software application that is created in a single block. This application runs as a single process. It can only be deployed in unison, although multiple identical copies can be created.

2. **What problems can monoliths run into?**

   As they grow, monoliths can experience the following problems:

- The code becomes too big and difficult to read.
- Scalability problems.
- The need to coordinate deployments.
- Bad usage of resources.
- It's not possible to use conflicting technologies for different situations (for example, different versions of the same library, or two programming languages).
- A single bug and deployment can affect the whole system.

3. **Can you describe the microservice architecture?**

   The microservice architecture is a collection of loosely coupled specialized services that work in unison to provide a comprehensive service.

4. **What is the most important property of a microservice?**

   The most important property of a microservice is that they can be deployed independently so that they can be developed independently.

5. **What are the main challenges that we need to overcome when migrating from a monolith to a microservice?**

   Possible challenges include the following:

   - Big changes are needed that require us to change the way the services operate, including the culture of the teams. This can result in training, which is costly.
   - Debugging a distributed system is more complicated.
   - We need to plan changes so that they don't interrupt the service.
   - There's a significant amount of overhead for each microservice that's developed.
   - We need to find a balance between allowing each team to decide how to work and standardizing to avoid reinventing the wheel.
   - We need to document the service so that we can interact with another team.

6. **How can we make such a migration?**

   We need to analyze the system, measure, plan accordingly, and execute the plan.

7. **Describe how we can use a load balancer to migrate from an old server to a new one, without interrupting the system.**

   First, we have to configure the load balancer so that it points to the old web server, which will make the traffic cross through the web server. Then, we have to change the DNS record so that it points to the load balancer. After the traffic has gone through the load balancer, we need to create a new entry for the new service so that the load balancer splits the traffic between both. After confirming that everything works as expected, we need to remove the entry from the old service. Now, all the traffic will be routed to the new service.

Chapter 2

1. **What are the characteristics of RESTful applications?**

   While a RESTful application is understood as a web interface that makes URIs into object representations and manipulates them through HTTP methods (and normally formats the requests with JSON), the textbook characteristics of a REST architecture are as follows:

- Uniform interface
- Client-server
- Stateless
- Cacheable
- Layered system
- Code on demand (optional)

   You can find out more about the REST architecture at `https://restfulapi.net/`.

2. **What are the advantages of using Flask-RESTPlus?**

   Some of the advantages of using Flask-RESTPlus are as follows:

- Automatic Swagger generation.
- A framework that can define and parse inputs and marshal outputs.
- It allows us to structure the code in namespaces.

3. **What are some alternatives to Flask-RESTPlus?**

   Other alternatives include Flask-RESTful (this is similar to Flask-RESTPlus, but it doesn't support Swagger) and the Django REST framework, which has a rich ecosystem that's full of third-party extensions.

4. Name the Python package used through the tests to fix the time.

   `freezegun`.

5. **Describe the authentication flow.**

   An encoded token is generated by the authentication system (the User Backend). This token is encoded with a private key that only the User Backend has. This token is encoded in JWT and contains a user ID, as well as other parameters that, for example, tell us how long the token is valid for. This token is included in the `Authentication` header.

   The token is obtained from the header and decoded using the corresponding public key, which is stored in the Thoughts Backend. This allows us to obtain the user ID independently with the certainty that it has been validated by the User Backend.

6. **Why did we choose SQLAlchemy as a database interface?**

   SQLAlchemy is well supported in Flask and allows us to define already existing databases. It is highly configurable and allows us to work at a low level, that is, near the underlying SQL, and at a higher level, which removes the need for any boilerplate code. In our use case, we inherited a database from the legacy system, thereby making the need to work seamlessly with an existing schema a necessity.

Chapter 3

1. **What does the FROM keyword do in a Dockerfile?**

   It starts our image from an existing one, adding more layers to it.

2. **How would you start a container with its predefined command?**

   You would run the following command:

   ```
   docker run image
   ```

3. **Why won't creating a step to remove files from a Dockerfile create a smaller image?**

   Due to the layered structure of the filesystem that's used by Docker, each step in a Docker file creates a new layer. The filesystem is the result of all the operations working in tandem. The final image includes all the existing layers; adding a layer never reduces the size of the image. A new step for deleting will not be present in the final image, but it will always be available as part of the previous layer.

4. **How does a multistage Dockerfile work?**

   A multistage Dockerfile contains more than one stage, each of which will start with a `FROM` command, which specifies the image that acts as the starting point. Data can be generated in one stage and then copied to another.

   Multistage builds are useful if we wish to reduce the size of the final images; only the resulting data will be copied to the final stage.

5. **What is the difference between the run and exec commands?**

   The `run` command starts a new container from an image, while the `exec` command connects to an already existing running container. Note that if the container stops while you are executing it, the session will be closed.

   Stopping a container can occur in an `exec` session. The main process that is keeping the container running is the `run` command. If you kill the command or stop it in any other way, the container will stop and the session will be closed.

6. **When should we use the -it flag?**

   When you need to keep a Terminal open, for example, to run `bash` commands interactively. Remember the mnemonic *interactive Terminal*.

7. **What are the alternatives to using uWSGI to serve web Python applications?**

   Any web server that supports the WSGI web protocol can be used as an alternative. The most popular alternatives are Gunicorn, which aims to be easy to use and configure, `mod_wsgi`, an extension of the popular Apache web server that supports WSGI Python modules, and CherryPy, which includes its own web framework.

8. **What is docker-compose used for?**

   `docker-compose` allows for easy orchestration, that is, we can coordinate several interconnected Docker containers so that they work in unison. It also helps us configure Docker commands since we can use the `docker-compose.yaml` file to store the configuration parameters for all the affected containers.

9. **Can you describe what a Docker tag is?**

   A Docker tag is a way of labeling images while keeping their root names. It typically marks different versions of the same application or software. By default, the `latest` tag will be applied to an image build.

10. **Why do we need to push images to a remote registry?**

    We push images to a remote registry so that we can share images with other systems and other developers. Docker builds images locally unless they need to be pushed to another repository so that other Docker services can use them.

Chapter 4

1. **Does increasing the number of deployments reduce the quality of deployments?**

   No; it has been shown that increasing the number of deployments has a strong correlation with their quality increasing. A system that is capable of deploying quickly has to rely on strong automated tests, which increases the stability and overall quality of the system.

2. **What is a pipeline?**

   A pipeline is an ordered succession of steps or stages that are used to perform a build. If one of the steps fails, the build stops. The order of steps should aim to maximize the early detection of problems.

3. **How do we know if our main branch can be deployed?**

   If we automatically run our pipeline to generate a build on each commit, we should detect problems on the main branch as soon as they are committed. The build should assure us that the top commit of the main branch can be deployed. A breakage in the main branch should be fixed as soon as possible.

4. **What is the main configuration source for Travis CI?**

   The `.travis.yml` file, which can be found in the root directory of the repository.

5. **By default, when does Travis CI send notification emails?**

   Travis CI sends notification emails whenever a build breaks and when a previously broken branch passes successfully. Successful builds occur when the previous commit was successful but not reported.

6. **How can we avoid merging a broken branch into our main branch?**

   We can avoid this by configuring in GitHub, which ensures that the branch passes the build before we merge it into a protected branch. To ensure that the feature branch has not deviated from the main one, we need to force it to merge with a build. It needs to be up to date with the main branch for this to happen.

7. **Why should we avoid storing secrets in a Git repository?**

   Due to the way Git works, any secret that's introduced can be retrieved by looking at the commit history, even if it's been removed. Since the commit history can be replicated in any cloned repository, it makes it impossible for us to verify whether it's correct—we can't rewrite the commit history into a cloned repository. Secrets should not be stored in Git repositories unless they are properly encrypted. Any secret that's stored by mistake should be removed.

Chapter 5

1. **What is a container orchestrator?**

   A container orchestrator is a system where we can deploy multiple containers that work in unison, as well as manage provisioning and deployment in an ordered fashion.

2. **In Kubernetes, what is a node?**

   A node is a physical server or virtual machine that is part of the cluster. Nodes can be added or removed from the cluster and Kubernetes will migrate or restart the running containers accordingly.

3. **What is the difference between a pod and a container?**

   A pod can contain several containers that share the same IP. To deploy a container in Kubernetes, we need to associate it with a pod.

4. **What is the difference between a job and a pod?**

   A pod is expected to run constantly. A job, or cron job, performs a single action, and then all the pod containers finish their execution.

5. **When should we add an Ingress?**

   We should add an Ingress when we need to be able to access a service externally from the cluster.

6. **What is a namespace?**

   A namespace is a virtual cluster. All of the definitions that are inside a cluster need to have unique names.

7. **How can we define a Kubernetes element in a file?**

   We need to specify it in YAML format and provide information about its API version, the kind of element it is, a metadata section with a name and namespace, and the element's definition in a `spec` section.

8. **What is the difference between the kubectl get and describe commands?**

   `kubectl get` obtains several elements, such as services or pods, and displays their basic information. `describe`, on the other hand, accesses a single element and presents much more information about it.

9. **What does the CrashLoopBackOff error indicate?**

   This error indicates that a container has finished executing the defined starting command. This error only occurs in relation to pods since they're never supposed to stop executing.

Chapter 6

1. **What are the three microservices that we are deploying?**

   The following are the three microservices that we are deploying:

- The Users Backend, which controls authentication and how users are handled.
- The Thoughts Backend, which stores thoughts and allows us to create and search for them.

- The Frontend, which provides us with a user interface so that we can interact with the system. It calls the other two microservices through RESTful calls.

2. **Which of the three microservices requires the other two microservices to be available?**

   The Frontend calls the other two microservices, so they need to be available for the Frontend to work.

3. **Why do we need to use external IPs to connect to microservices while they're running in docker-compose?**

   `docker-compose` creates an internal network for each microservice, so they need to communicate using an external IP so that they're routed properly. While we're exposing ports in the host computer, the external host IP can be used.

4. **What are the main Kubernetes objects that are required for each application?**

   For each microservice, we provide a deployment (which automatically generates a pod), a service, and an Ingress.

5. **Are any of the objects not required?**

   An Ingress for the Users Backend and the Thoughts Backend isn't strictly required since they can be accessed through the node port, but it does make accessing them easier.

6. **Can we detect issues if we scale to more than one pod or any other microservice?**

   The Users Backend and Thoughts Backend creates a pod with two containers, which includes the database. If we create multiple pods, we will be creating multiple databases, and alternating between them can cause issues.

   For example, if we create a new thought in one of the pods, we won't be able to search for it if the request was made in another pod.

   On the other hand, the Frontend can be scaled with no issue.

7. **Why are we using the /etc/hosts file?**

   We are using this file so that we can define a `host` that routes to our local Kubernetes cluster. This avoids us having to define an FQDN and configure a DNS server.

Chapter 7

1. **Why shouldn't we manage our own Kubernetes cluster?**

   Since Kubernetes is an abstraction layer, it is more convenient to have a cloud provider take care of maintenance and management, as well as security best practices. It's also quite cheap to delegate clusters to an existing commercial cloud provider.

2. **Can you name some commercial cloud providers that have a managed Kubernetes solution?**

   Amazon Web Services, Google Cloud Services, Microsoft Azure, Digital Ocean, and IBM Cloud are all commercial cloud providers that have a managed Kubernetes solution.

3. **What action do you need to perform to be able to push to an AWS Docker registry?**

   You need to log in to your Docker daemon. You can obtain the login command by using the following code:

   ```
   $ aws ecr get-login --no-include-email
   ```

4. **What tool do we use to set up an EKS cluster?**

   `eksctl` allows us to create the whole cluster from the command line, as well as scale it up or down as required.

5. **What are the main changes we made in this chapter so that we could use the YAML files from the previous chapters?**

   We had to change the image definition to be able to use the AWS registries. We included the liveness and readiness probes, as well as a deployment strategy.

These are only added to the `frontend` deployment. Adding the rest of the deployments is left to you as an exercise.

6. **Are there any Kubernetes elements that are not required in this cluster?**

   The Ingress element isn't strictly required since the Thoughts Backend and the Users Backend can't be accessed externally. The frontend service is capable of creating an externally facing ELB.

   Don't feel like you're limited by our configuration. You can manually configure an ELB so that you can access the cluster in different ways, so you can use the Ingress configuration if you wish.

7. **Why do we need to control the DNS associated with an SSL certificate?**

   We need to prove that we own the DNS so that the SSL certificate can verify that only legitimate owners of the DNS address have access to a certificate for that DNS. This is the root element of HTTPS and states that you are communicating privately with the owner of a particular DNS.

8. **What is the difference between the liveness and the readiness probes?**

   If the readiness probe fails, the pod won't accept requests until it passes again. If the liveness probe fails, the container will be restarted.

9. **Why are rolling updates important in production environments?**

   They are important as they avoid interruptions in the service. They add workers one by one while removing old ones, ensuring that the number of available workers at any time remains the same.

10. **What is the difference between autoscaling pods and nodes?**

    Since nodes are reflected in physical instances, scaling them affects the resources that are in the system. Meanwhile, scaling pods uses the resources that are available to them, but doesn't modify them.

    In other words, increasing the number of nodes we have adds more hardware that needs to be run on the system. This has a cost associated with it in that we need to hire more hardware from our cloud provider. Increasing the number of pods we have doesn't have a cost in terms of hardware, which is why there should be some overhead to allow for increases.

    Both strategies should be coordinated so that we can react quickly to load increases and, at the same time, reduce the amount of hardware that's being utilized so that we can reduce costs.

11. **In this chapter, we deployed our own database containers. In production, this isn't required. However, if you're connected to an already existing database, how would you do this?**

    The first step would be to change the environment variables in the `thoughts_backend/deployment.yaml` and `users_backend/deployment.yaml` files. The main one to connect to is `POSTGRES_HOST`, but the user and password will probably have to change as well.

    Instead of connecting to `POSTGRES_HOST` as an IP or DNS address directly, we could create an internal Kubernetes service called `postgres-db` that points toward an external address. This could help us abstract the address of the external database.

    This would be deployed in one go to ensure that we can connect to the external database.

    Then, we can remove the database containers that were described in the deployments, that is, `thoughts-backend-db` and `users-backend-db`. The images for these containers are only used for testing and development.

Chapter 8

1. **What is the difference between using a script to push new code to servers and using a configuration management tool such as Puppet?**

    When using a script to push new code to servers, every server needs to have the code pushed individually. *Puppet* and other configuration management tools have a centralized server that receives new data and distributes it appropriately. They also monitor the fact that the servers are running as expected and can perform remediation tasks.

    Configuration management tools are used for big clusters since they reduce the amount of work that needs to be handled in custom scripts.

2. **What is the core idea behind DevOps?**

    The core idea behind DevOps is to empower teams so that they have control over performing their own deployments and their infrastructure. This requires a safety network in the form of automated procedures to ensure that these operations are easy, safe, and quick to perform.

3. **What are the advantages of using GitOps?**

   The main advantages of using GitOps are as follows:

- Git is a common tool that most teams already know how to use.
- It keeps a copy of the infrastructure definition, which means we can use it as a backup and recover from catastrophic failures or create a new cluster based on a previous definition with ease.
- The infrastructure changes are versioned, which means we can make small discrete changes one by one and revert any of them if there's a problem.

4. **Can only GitOps be used in Kubernetes clusters?**

   Though GitOps does have synergy with Kubernetes, since Kubernetes can be controlled by YAML files, there's nothing stopping us from using a Git repository to control a cluster.

5. **Where does the Flux deployment live?**

   It lives in its own Kubernetes cluster so that it can pull data from Git.

6. **What do you need to configure in GitHub so that Flux can access it?**

   You need to add an SSH key to the deployment keys of the GitHub repository. You can obtain an SSH key by calling `fluxctl identity`.

7. **When you're working in production environments, what features that are provided by GitHub ensure that we have control over deployments?**

   We need to have review and approval before we can merge into the main branch, which triggers a deployment. The inclusion of code owners to force approval from specific users can help us control delicate areas.

Chapter 9

1. **When new business features are received, what analysis needs to be done in a system working under a microservice architecture?**

   We need to determine what microservice or microservices the new business feature affects. A feature that affects multiple microservices makes its implementation more difficult.

2. **If a feature requires two or more microservices to be changed, how do we decide which should be changed first?**

   This should be done in a back-to-forward way in order to keep backward compatibility. New features should be added while keeping backward compatibility in mind, so the possibilities are limited. Once the backend is ready, the frontend can be changed accordingly so that we can take advantage of the new feature.

3. **How does Kubernetes help us set up multiple environments?**

   Creating new namespaces is very easy in Kubernetes. Since the definition of the system is encapsulated in YAML files, they can be copied and modified to create a duplicate environment. This can be used as a baseline and then evolved.

4. **How do code reviews work?**

   The code in one branch is compared with the main branch. Another developer can look at the differences between them and make comments, asking for clarification or changes. These can then be discussed and then the code can be approved if the reviewer thinks it's good enough. The merge can be blocked until it has one or more approvals.

5. **What is the main bottleneck for code reviews?**

   The main bottleneck is not having a reviewer to provide feedback and approve the code. That's why it's important to have enough people that can perform the role of a reviewer.

6. **Under GitOps principles, are the reviews for deployment different from code reviews?**

   No; under GitOps, the deployments are treated as code, so they can be reviewed just like any other code review.

7. **Why is it important to have a clear path to deployment once a feature is ready to be merged into the main branch?**

   It's important to have a clear path to deployment so that everyone is on the same page. It also provides a clear expectation of how fast deployments can happen. By doing this, we can specify when a review is required.

8. **Why are database migrations different from regular code deployments?**

   They are different because they can't be rolled back easily. While a code deployment can be rolled back so that the previous image is deployed again, database migrations make changes to the schema of the database or data, which may cause data loss if they're reverted. Normally, database migrations are forward-only, and any problems that occur need to be corrected with a new deployment.

   This is the main reason why we have to take extra care with database migrations and ensure they are not backward incompatible.

Chapter 10

1. **What is the observability of a system?**

   It's the capacity of a system. It lets you know what its internal state is.

2. **What are the different severity levels that are available in logs by default?**

   In order of increasing severity, the different severity levels are `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

3. **What are metrics used for?**

   Metrics allow you to find out the aggregated statuses of the events that are occurring on the system and allow you to understand the general state of the system.

4. **Why do you need to add a request ID to the logs?**

   You need to add a request ID to the logs so that you can group all of the logs that correspond to the same request.

5. **What kinds of metrics are available in Prometheus?**

   Counters, which count a particular event; gauges, which keep track of a value that can go either up or down; and histograms (or summaries), which track events with a value associated with them, such as the times that events occur or when the status codes of requests are being returned.

6. **What is the 75th percentile in a metric and how is it different from the average?**

   For histograms, the 75^{th} percentile is where *25%* of the events are higher than the average, while *75%* are lower than it. The average is found by adding all of the values together and dividing that value by the number of values that were added together initially. Typically, the average will be close to the 50th percentile, although this depends on how the values are distributed.

   The 90^{th}-95^{th} percentile is good if we wish to determine latency since it provides upper times for requests, not counting outliers. The average can be skewed by outliers and thus not provide a realistic number for the vast majority of requests.

7. **What are the four golden signals?**

   The four golden signals are four measurements that gather a description of a system's health. They are the latency of requests, the amount of traffic, the percentage of returned errors, and the saturation of resources.

Chapter 11

1. **What are the differences between releasing changes in a microservice architecture system and a monolith?**

   Releasing a change in a monolith will only involve one repository since the monolith is only one code base. Some changes that are made in a microservice architecture will need us to change two or more microservices so that we can allocate them. This requires more planning and care since we need to ensure that this is properly coordinated. In a properly architecture microservice system, such multirepository changes should be relatively rare since they incur overhead.

2. **Why should release changes be small in a microservice architecture?**

   The advantage of microservices is that we can release microservices in parallel, which is quicker than a monolith release. However, given that a release in a microservice could potentially affect other microservices, they should work in an iterative way, reducing the size of the changes and increasing their rate of deployment.

   A small change is less risky and easier to roll back if that's required.

3. **How does semantic versioning work?**

   In semantic versioning, versions have three numbers: a *Major* version number, a *Minor* version number, and a *Patch* version number. These are all separated by dots:

- An increase in a Patch version only fixes bugs and security problems.
- An increase in a Minor version adds more features, but without backward-incompatible changes.
- An increase in a Major version produces backward-incompatible changes.

4. **What are the problems with semantic versioning for internal interfaces in a microservice architecture system?**

   Since deployments in microservices are very common, and backward compatibility is very important, the meaning of a *major* release becomes diluted. Also, most of the consumers of the microservices are internal, so implicit communication between versions is less important.

   When releases become common, semantic versioning loses meaning since the objective is to continuously refine and improve the product, instead of marking big releases.

5. **What are the advantages of adding a version endpoint?**

   Any consumer that uses a microservice can request its version in the same way they can make any other request: by using a RESTful call.

6. **How can we fix the dependency problem in this chapter's code?**

   The code in this chapter has a dependency problem between TO BE FILLED.

7. **What configuration variables should we store in a shared ConfigMap?**

   We should store the configuration variables that are accessed by multiple microservices. Preemptively, we should store most of the configuration variables so that they can be reused.

8. **Describe the advantages and disadvantages of putting all the configuration variables into a single shared ConfigMap.**

   A single shared ConfigMap makes the configuration variables very explicit. It encourages everyone to reuse them and tell others what the configuration is used for in other microservices.

Changing the dependency of a microservice will trigger a restart, so changing a ConfigMap that acts as a dependency for everything will cause all the microservices in the cluster to restart, which is time-consuming.

Also, a single ConfigMap file can end up being quite big, and splitting it into several smaller ones can help us organize the data more efficiently.

9. **What's the difference between a Kubernetes ConfigMap and a Kubernetes Secret?**

   Kubernetes Secrets are better protected against unintentional access. The direct access tools won't display the Secret in plain text. Access to Secrets also needs to be configured in a more explicit way. On the other hand, ConfigMaps can be configured in bulk, so a pod will be able to access all of the values that have been stored in the ConfigMap.

10. **How can we change a Kubernetes Secret?**

    We can change a Secret using `kubectl edit`, but it will need to be encoded in Base64 format.

    For example, to replace the `postgres-password` secret with the `someotherpassword` value, we can use the following code:

```
$ echo someotherpassword | base64
c29tZW90aGVycGFzc3dvcmQK
$ kubectl edit secrets -n example thoughts-secrets
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while
saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
    postgres-password: c29tZW90aGVycGFzc3dvcmQK
...
secret/thoughts-secrets edited
```

    Once restarted, our pods will be able to use the new Secret.

11. **Imagine that, based on our configuration, we decided to change public_key.pub from a Secret to a ConfigMap. What changes do we have to make?**

    We need to change the ConfigMap so that it includes the file in `configuration.yaml`:

```
THOUGHTS_BACKEND_URL: http://thoughts-service
public_key.pub: |
  -----BEGIN PUBLIC KEY-----
  <public key>
  -----END PUBLIC KEY-----
USER_BACKEND_URL: http://users-service
```

    Note the indentation to delimitate the file. The `|` char marks a multiline string.

    Then, in the `deployment.yaml` file, we need to change the source of the mount from a Secret to a ConfigMap:

```
volumes:
    - name: public-key
      configMap:
          name: shared-config
          items:
              - key: public_key.pub
                path: public_key.pub
```

    Remember to apply these changes to the ConfigMap first so that they are available when the deployment files are applied.

    Note that this method creates an environment variable called `public_key.pub`, along with the content of the file, since it is applied as part of the `shared-config` ConfigMap. An alternative is to create an independent ConfigMap.

    The Secret can be deleted after all the pods have been restarted.

Chapter 12

1. **Why is a leading architect convenient for a microservice architecture system?**

   Structuring a system in a microservice architecture allows us to create independent services that can be handled in parallel. These services still need to communicate with each other and cooperate.

   Independent teams usually don't have a grasp of the big picture and tend to focus on their own projects. To help with the coordination and evolution of the system as a whole, independent teams need a leading architect who has a high-level overview of the system.

2. **What is Conway's Law?**

   Conway's Law is an adage that says that software structures replicate the communication structures of the organization that writes it.

   This implies that, to change the way the software is structured, the organization needs to change, which is a much more difficult task.

   To successfully design and evolve big systems, the organization needs to be taken into account and planned accordingly.

3. **How is technical debt introduced?**

   There are a lot of ways in which technical debt can be created.

   Normally, technical debt falls into one of the following four categories or a mix of them:

- By developing too quickly without taking the time to analyze other options
- By making a compromise to shorten development time while knowing that the comprise will need to be fixed later
- By not having a good enough understanding of the current system or tools, or having a lack of training or expertise
- By making incorrect assumptions regarding an external problem, thereby designing for something that doesn't necessarily need to be fixed

4. **Why is it important to create a culture so that we can work continuously to reduce technical debt?**

   It's important to create a culture so that we can avoid *software rot*, which is the continuous decay of performance and reliability due to adding complexity to existing software. Unless addressing technical debt becomes a continuous process, the day-to-day pressure of making new releases means we won't be able to perform maintenance.

5. **Why is it important to document problems in releases and share them with the rest of the team?**

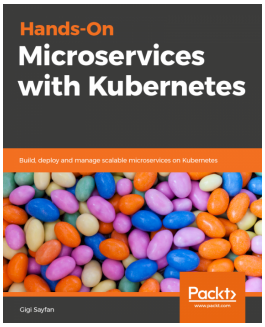
   This is important since every team can learn from the experience and solutions of others and improve their processes. This can also create an open culture where people aren't afraid of taking responsibility for their mistakes.

6. **What is the main objective of a post-mortem meeting?**

   The main objective of a post-mortem meeting is to create follow-up tasks that fix the causes of an incident. To do so, we need to be as confident as possible that the root cause has been successfully detected (this is also the secondary objective).

Other Books You May Enjoy

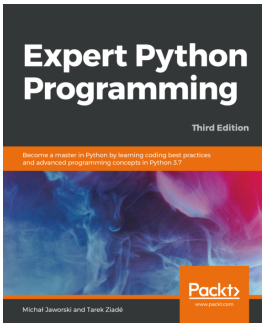
If you enjoyed this book, you may be interested in these other books by Packt:



Hands-On Microservices with Kubernetes
Gigi Sayfan

ISBN: 978-1-78980-546-8

- Understand the synergy between Kubernetes and microservices
- Create a complete CI/CD pipeline for your microservices on Kubernetes
- Develop microservices on Kubernetes with the Go kit framework using best practices
- Manage and monitor your system using Kubernetes and open source tools
- Expose your services through REST and gRPC APIs



Expert Python Programming - Third Edition
Tarek Ziadé, Michał Jaworski

ISBN: 978-1-78980-889-6

- Explore modern ways of setting up repeatable and consistent development environments
- Package Python code effectively for community and production use
- Learn modern syntax elements of Python programming such as f-strings, enums, and lambda functions
- Demystify metaprogramming in Python with metaclasses
- Write concurrent code in Python
- Extend Python with code written in different languages
- Integrate Python with code written in different languages

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W