‹packt›

1ST EDITION

Hands-On Microservices with Django

Build cloud-native and reactive applications with Python using Django 5

TIEME WOLDMAN

Hands-On Microservices with Django

Build cloud-native and reactive applications with Python using Django 5

Tieme Woldman

Hands-On Microservices with Django

Group Product Manager: Rohit Rajkumar

Publishing Product Manager: Bhavya Rao

Book Project Manager: Shagun Saini

Senior Editor: Nathanya Dias

Technical Editor: Simran Ali

Copy Editor: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Prashant Ghare

DevRel Marketing Coordinators: Anamika Singh and Nivedita Pandey

First published: April 2024

Production reference: 1010424

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83546-852-4

www.packtpub.com

Books are written by teams, not just by the author whose name is on the front cover. So, a big compliment to all the people who helped me to create this book.

– Tieme Woldman

Contributors

About the author

Tieme Woldman works as a freelance Python developer and technical writer. As a Python developer, he builds web and data engineering applications with Django and Python data transformation packages such as pandas. As a technical writer, he has written software and user documentation for software companies such as Instruqt, Noldus Information Technology, and Rulecube.

Tieme lives in the Netherlands, has a bachelor's degree in computer science, and holds several (technical) writing certifications.

About the reviewers

Uwem Akpabot has 13 years of progressive professional experience as a full stack software developer. He earned a master's degree in software engineering from the University of Greenwich, United Kingdom, in 2013. He has held managerial/supervisory positions where he led teams in the successful delivery of projects. His expertise includes Python programming, Java, PHP, JavaScript, and problem-solving with data structures and algorithms. As a developer, he is eager to create reliable solutions that are easy to use or modify and can perform flawlessly. As a gifted instructor, he creates excellent courses that help others learn programming.

Afeez Lawal is an experienced Django backend engineer with a proven track record of designing and developing innovative software solutions. With a keen eye for detail and a passion for problem-solving, Afeez has successfully delivered a range of projects, demonstrating expertise in backend development, particularly using the Django framework.

Table of Contents

8

Deploying Microservices with Docker 171

Part 3: Taking Microservices to the Production Level

9

Securing Microservices 191

10

Improving Microservices Performance with Caching 205

11

Best Practices for Microservices 221

12

Transforming a Monolithic Web Application into a Microservices Version 235

Preface

Hello, fellow Django developers and others interested in enhancing web applications with microservices. **Microservices** are playing an important role in today's web applications. In 2020 alone, market researchers from Gartner saw a 42 percent increase in mentions of *microservices architecture* on social media. So, microservices are hot, raising questions regarding topics such as what they are and how we can apply them.

Fortunately, there is now enough experience building microservices to answer these questions from a real-world perspective. To give you a first idea, a microservice is a single-task piece of software that is part of a larger application and can be used by different applications.

As a result, microservices run asynchronously while collectively completing a process. As such, microservices are the opposite of monolithic applications, which perform all tasks sequentially from a single program, whereby one task can freeze the user experience because it takes seconds to complete.

This created the need for asynchronous processing with microservices as they let us split applications into task-driven components that run independently. So, the application proceeds without delay, which improves the user experience.

To develop and implement microservices, we need these main parts:

- A producer, which is a program that offloads a task to a microservice
- A task queue manager that passes the tasks to the microservices
- A microservice, which is a program that listens to a queue and executes when a task arrives

There are many ways to implement these parts. Producers can be Django apps or other applications, such as React and Vue components. For the task queue manager, we have a choice of systems, such as Redis and RabbitMQ. And we can develop microservices in Python or another programming language, such as Node.js.

This book focuses on developing Django microservices and, therefore, covers these choices for the main microservices parts:

- Django apps as producers.
- Celery and Redis as task queue managers, as these are most common for Django. But the book also covers RabbitMQ, as this gives more profound insight into task queuing.
- Python as the programming language for developing microservices.

The format of this book is hands-on, meaning it provides you with the necessary information about concepts and then provides extensive practical steps, examples, and explanations to build Django microservices yourself.

Who this book is for

This book is for Django developers who want to take the next step in backend application development by creating advanced applications with cloud-native microservices. Backend developers with working knowledge of Flask or other Python web frameworks would also benefit.

What this book covers

Chapter 1, *What Is a Microservice?*, provides an overview of the microservices architecture, its components, and its benefits. It also covers an approach to designing microservices.

Chapter 2, *Introducing the Django Microservices Architecture*, provides insight into the Django microservices architecture and walks through creating an example microservice to see what it takes to build microservices.

Chapter 3, *Setting Up the Development and Runtime Environment*, provides an overview of the software, systems, and packages needed to build and run microservices. It also covers the steps to install and configure the required parts.

Chapter 4, *Cloud-Native Data Processing with MongoDB*, lays a data foundation for a microservices application with the cloud version of the popular NoSQL database, MongoDB. This includes mapping CRUD operations to HTTP methods.

Chapter 5, *Creating RESTful APIs for Microservices*, extends the data layer for Django microservices by explaining how to create a RESTful API for serving microservice data operations toward MongoDB.

Chapter 6, *Orchestrating Microservices with Celery and RabbitMQ*, provides the basics of the task queue managers, Celery and RabbitMQ. It also explains how to build microservices and monitor tasks.

Chapter 7, *Testing Microservices*, provides complementary approaches for testing individual microservices and the entire microservice application. It also shows how to test a microservices application with Selenium automatically.

Chapter 8, *Deploying Microservices with Docker*, provides an overview of containerizing microservices with Docker and its benefits. It also demonstrates how to containerize microservices and set up a deployment cycle.

Chapter 9, *Securing Microservices*, shows how to secure microservices both from calling clients and microservices calling each other.

Chapter 10, *Improving Microservices Performance with Caching*, explains and demonstrates caching to maintain and improve microservices' performance when user demand increases. Both Django's built-in caching framework and Redis will be covered, including fully working samples.

Chapter 11, *Best Practices for Microservices*, provides hands-on tips and advice for optimally maintaining and running microservices. It addresses topics such as error handling, logging, and documenting.

Chapter 12, *Transforming a Monolithic Web Application into a Microservices Version*, walks through transforming an existing monolithic web application into a microservices version, as many Django developers will face this scenario.

To get the most out of this book

You need to have experience in developing Django applications. You don't have to be a Django veteran with more than a decade of experience, but you should be able to create a Django application with forms. If you can create class-based forms and pages, it's totally fine. A working knowledge of Flask or other Python web frameworks is also proficient. Furthermore, a basic understanding of Web APIs will help.

Software/hardware covered in the book	**Operating system requirements**
Django	Windows, macOS, or Linux
Redis	
RabbitMQ	
Celery	
MongoDB Cloud version	

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main`. If there's an update to the code, it will be updated in the 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!

Conventions used

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

`Code in text`: 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 request applies the `GET` request to access the `api.trip.com` endpoint and addresses the `hostels` resource."

A block of code is set as follows:

```
class AddressViewSet(viewsets.ModelViewSet):
    queryset = Address.objects.all()
    serializer_class = AddressSerializer
```

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

```
1 from rest_framework import generics
2 from .models import Address
3 from .serializers import AddressSerializer
4
5 class AddressList(generics.ListCreateAPIView):
6    queryset = Address.objects.all()
7    serializer_class = AddressSerializer
8
9 class AddressDetail(generics.
                      RetrieveUpdateDestroyAPIView):
10    queryset = Address.objects.all()
11    serializer_class = AddressSerializer
```

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

```
$ curl -d '{"hostel_id":24, "start":"2024/03/01", "end":"2024/03/06"}'
-H "Content-Type: application/json" -X POST http://api.trip/v1/
hostels/
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "In the **Name** and **Address** fields, enter the values of your choice."

> **Tips or important notes**
>
> 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, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

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` and fill in the form.

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@packtpub.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`.

Share Your Thoughts

Once you've read *Hands-On Microservices with Django*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83546-852-4`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Introducing Microservices and Getting Started

In this part, you'll learn what microservices are and how you design them. Furthermore, you'll learn about the Django microservices architecture and its components. Finally, you'll master setting up your development and runtime environment for building and deploying Django microservices applications.

This part contains the following chapters:

- *Chapter 1*, *What Is a Microservice?*
- *Chapter 2*, *Introducing the Django Microservices Architecture*
- *Chapter 3*, *Setting Up the Development and Runtime Environment*

1
What Is a Microservice?

A **microservice** is a small, independent, and single-tasked piece of software that is part of a larger application and can be used by different applications. An important point to note is that a microservice is **independent**, meaning that if there is a problem with a particular microservice, that problem can't affect the application's overall functionality. You can modify and debug a microservice without affecting other microservices or the application itself.

Furthermore, a microservice is a self-contained program that runs asynchronously while collectively completing a process with other microservices or applications. For example, to reset a password on request, one microservice resets the login retry threshold, another sends an email with a reset link, and another writes a log message. They all act independently on a single task, but together, they cover the password reset process.

As such, microservices are the opposite of monolithic applications that perform all tasks sequentially from a single program, whereby one task can freeze the user experience because it takes multiple seconds to complete. This creates the need for asynchronous processing with microservices, as they let us split applications into task-driven components that run independently. So, an application can proceed without delay, which improves the user experience.

In this chapter, you'll learn about microservices and their components. Furthermore, you'll learn about the benefits of microservices to justify your transition effort. Finally, you'll master the design principles of microservices to build them yourself.

By the end of this chapter, you'll have a good understanding of what microservices are and know how to design a microservices-based application.

To achieve this, this chapter covers the following topics:

- Comparing monolithic web applications and microservices
- Exploring the microservices architecture and its components
- Listing the benefits of microservices
- Distinguishing microservice types
- Designing microservices

We have some exciting ground to cover, so let's start by exploring the differences between monolithic web applications and microservices.

Comparing monolithic web applications and microservices

To master a new concept such as microservices, contrasting it to a known concept can be insightful. Because microservices tackle the limits and drawbacks of monolithic applications, it's interesting to look at the differences between microservices and their monolithic opponents. So, let's focus on an example monolithic application and then compare it with a microservices version.

A monolithic version of a Discount Claim app

Imagine that the first 150 responders can claim a discount on a new and cool AI-powered code editor. The supplier of the code editor set up a web page on its website where responders can claim their discount. All they have to do is supply their name and email address, after which they receive an email with a discount code to order the code editor.

Technically, the web page is a monolithic single-page application that sequentially executes these tasks:

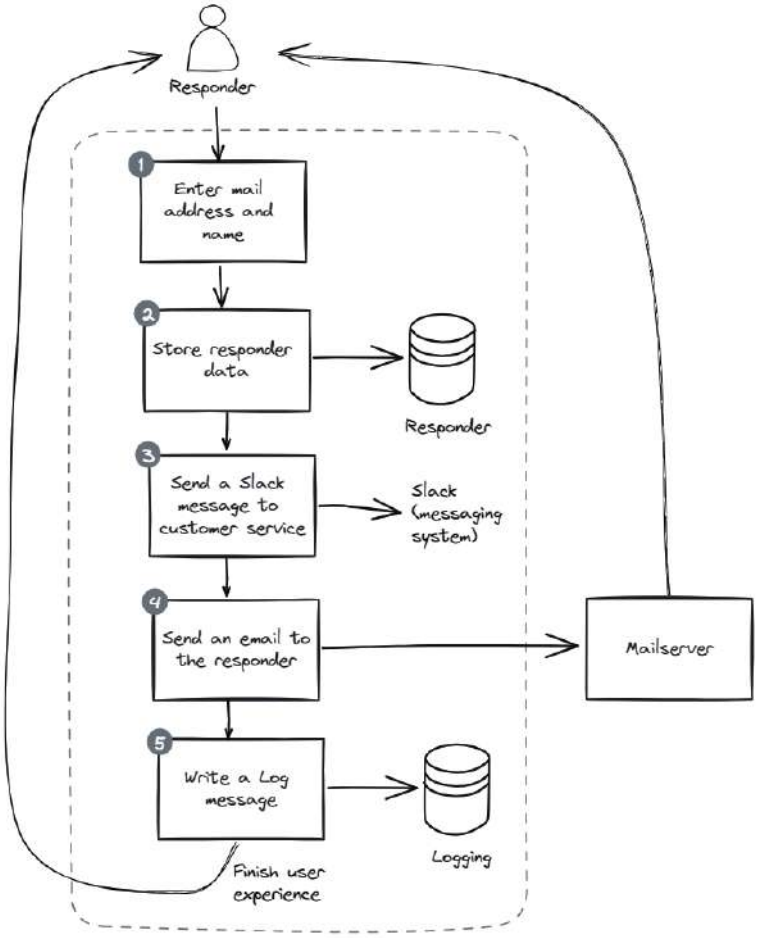

Figure 1.1 – Monolithic application architecture with synchronous tasks

As soon as the responder enters their data in the first task, the second task stores the responder's data in a database for later reference. Then, the third task sends an internal message to customer service, and then, the fourth task sends the email with the discount code to the responder. Finally, the last task writes a log message and finishes the user experience.

As such, the application works well, but the fourth task that sends the email takes almost 10 seconds to complete while freezing the user experience. The responders do receive their email, but they start joking about whether the code editor will be as slow as the claim page on social media.

A microservices version of the Discount Claim app

The supplier of the new code editor wants to prevent the social media posts from turning into negative publicity and consults their backend developer, who comes up with a microservices alternative:

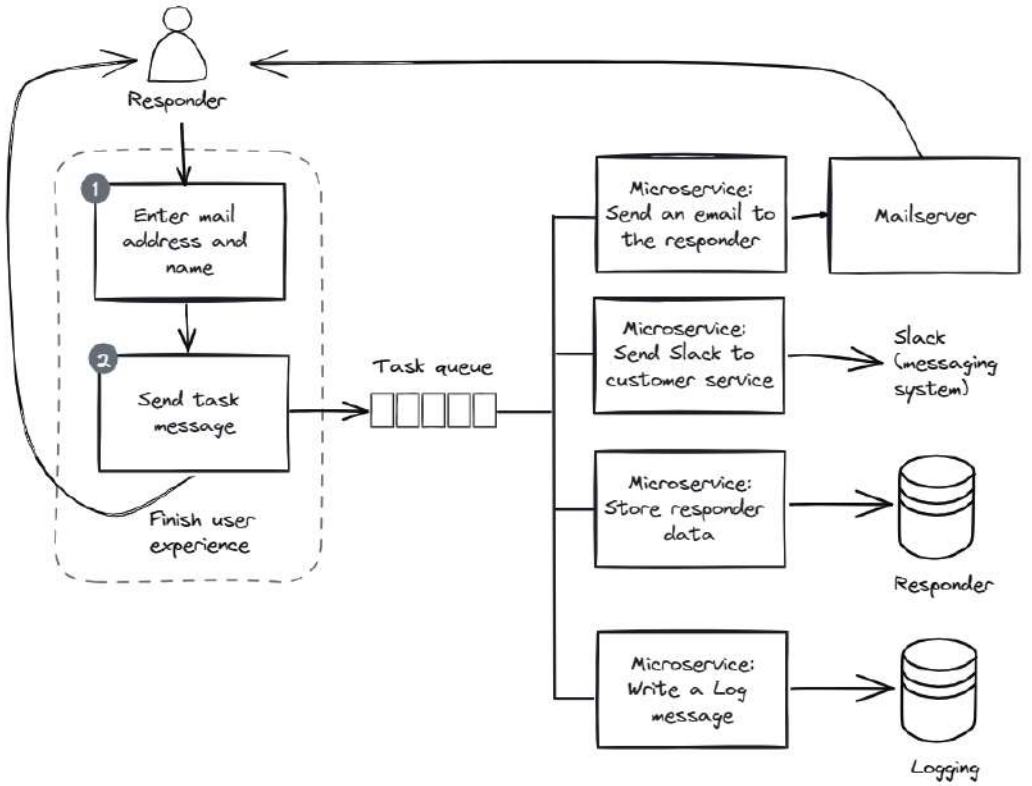

Figure 1.2 – Microservices architecture with asynchronous tasks

Now, when the responder enters their mail address and name, the application sends a message to a task queue and finishes the user experience instantly. Each microservice is an independent task that runs asynchronously in parallel and listens if a new message arrives in the queue. If so, the microservice parses the message and executes the accompanying task.

The order in which the microservices execute is irrelevant, as is their duration time. So, the microservice that sends the email doesn't hold up the other microservices and the user experience.

After implementing this microservices architecture, responders stopped joking and reacted positively on social media.

A Python microservice example

Microservices can be implemented in Python as a single-task program or script. To give you an idea of what they can look like, this is a microservice in Python that writes a log message whenever it detects a new task from the task queue:

```
1 from celery import Celery
2
3 app = Celery('log', broker='pyamqp://guest@localhost//')
4
5 @app.task
6 def write_log(log_message):
7     ...
```

Important note

For compactness, we do not format code blocks according to *autopep8* or any other style guide.

Line 1 imports `Celery`, the task queue manager.

Line 3 connects the microservice to the `log` task queue, so this microservice listens and receives every message sent to the queue. Furthermore, this line ensures that the microservice runs endlessly, until we stop it with *Ctrl + C*. We'll go into the parameters of this line in *Chapter 6*, *Orchestrating Microservices with Celery and RabbitMQ*.

Line 5 decorates the `write_log` function in *line 6* that writes a log message. This opens the function for incoming messages that contain the `log_message` argument to process. We will leave the implementation of the `write_log` function for now to keep this example simple.

Characteristics of microservices

Okay, now that we have contrasted microservices with monolithic applications and have seen an example of a microservice, let's sum up what this teaches us about microservices:

- Microservices are programs that execute a single task, such as sending an email or writing data to a database.
- Microservices are self-contained. They have everything on board to fulfill their task and only need a message to start them off.
- Microservices are independent – not only from other microservices but also from applications. So, different applications can call the same microservices.
- Microservices run asynchronously and execute, depending on their duration, in parallel.

- Microservices listen to a task queue for a new message, and if a message arrives, the microservices start executing.
- Microservices collectively execute a process. Although they run asynchronously with different durations, they ultimately perform a complete process, such as handling a discount request.

Excellent; the comparison of microservices and monolithic applications and the characteristics of microservices gives us a firm understanding of what they are. Now, let's look at the components that make up a microservice architecture to expand our knowledge of microservices.

Exploring the microservices architecture and its components

As we saw with the characteristics of microservices, microservices are independent programs that execute a task on command. If we focus on this, we will recognize the ideas of decentralization and specialization. Because instead of one big, centralized program, we have several small, decentralized programs that make microservices. And instead of having one program that does everything, we have multiple programs, each executing a specialized task.

Thus, the concepts of decentralization and specialization are the starting point of microservices. When we translate these concepts into a practical microservices architecture, we can see the components that constitute such an architecture.

Because, in addition to microservices, we need a mechanism that orchestrates the messages that fire microservices, such as the task queue we saw earlier. But we also need functionality in the second task of *Figure 1.2* that sends a task to the queue, and the microservices need functionality to listen to a queue and pick up a task.

Important note

The terms **task**/**message** are used interchangeably. However, *task* focuses a little more on functionality, and *message* focuses on technology.

In this book, we'll use **message** when we look at message queue brokers, such as RabbitMQ and Redis, and **task** when we dive into the task manager Celery.

Altogether, this brings us to this general microservices architecture or design pattern:

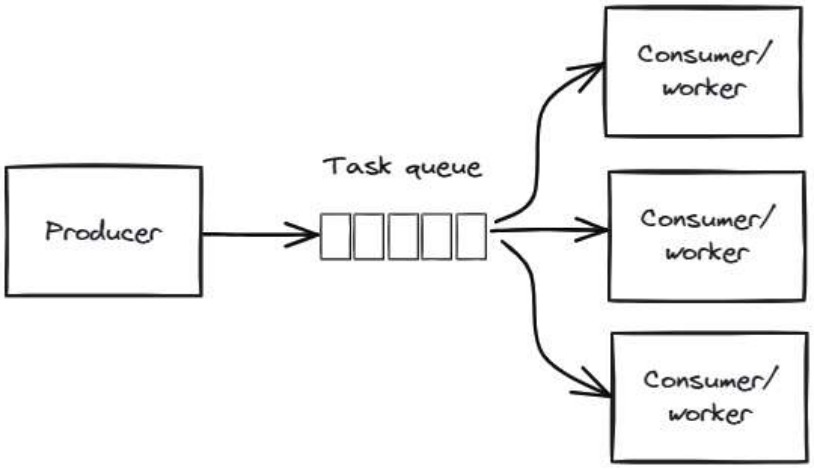

Figure 1.3 – A general microservices architecture with its components

This architecture introduces some known and unknown concepts; let's look at what they mean:

- **Producer**: A program or script that offloads tasks to a queue to be executed separately.
- **Task queue**: A broker that manages tasks between the producer and consumer. This includes distributing tasks over multiple consumers for load balancing.
- **Consumer**: A program or script that listens to a queue for new tasks and executes when a task arrives. Consumers are also called **workers** because they are the components that do the actual work.

If we go back to the microservice example of *Figure 1.2* and apply these microservices architecture concepts, we arrive at this overview:

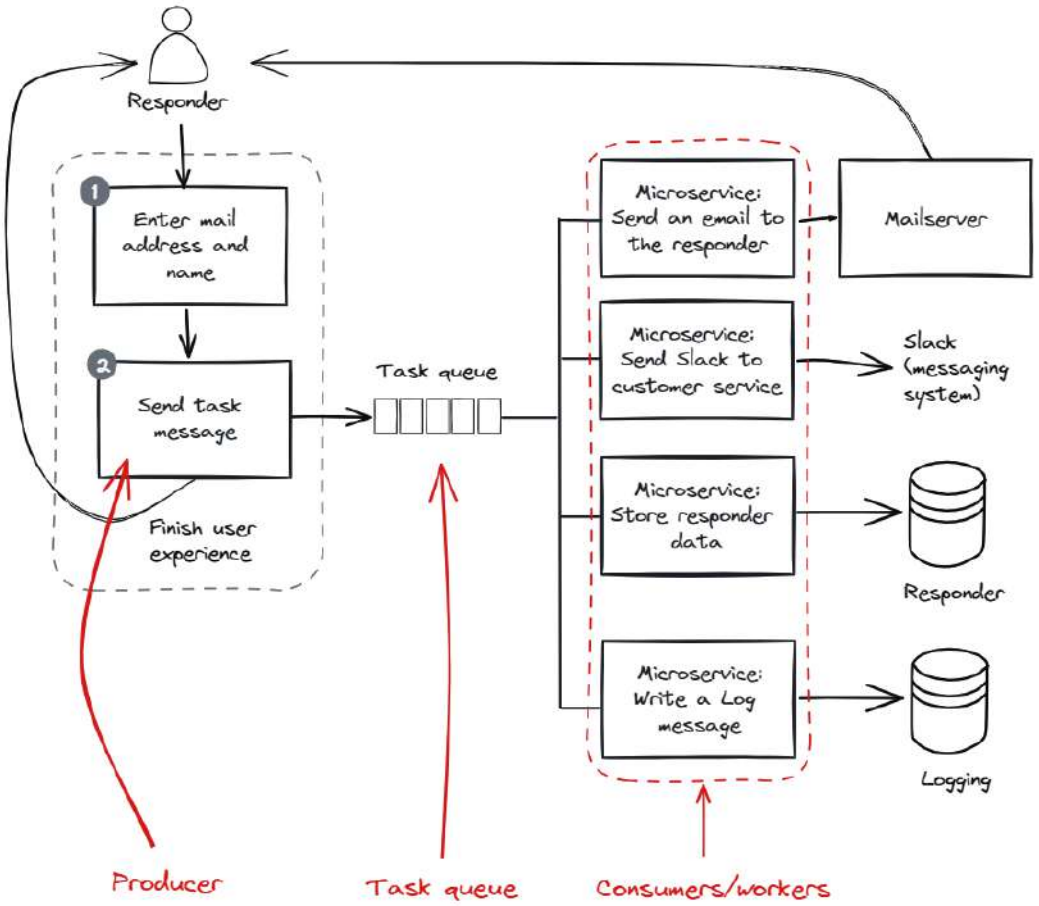

Figure 1.4 – An example microservices architecture with its components

So, task two is the *producer* because it sends and offloads the tasks, and the microservices are the *consumers/workers*, as they listen to new incoming tasks to execute. In between them, we find the *task queue*.

An analogy to deepen our understanding of microservices

Because you are a developer, you are familiar with abstract concepts, and therefore, you understand without question the concept of microservices architecture. But an analogy can deepen our understanding, so let's take a moment to hear this real-life story of how microservices work.

Meet Miquel – he works in finance, and in his free time, he enjoys cooking for friends. Everyone loves his diners, and his friends keep telling him he should do more with cooking. One day, he cycles to work and sees a beautiful property for sale – perfect for starting a small restaurant. To cut a long story short, he starts a restaurant and quits his job. He works alone in the kitchen as a chef, and a friend serves the customers. They get great reviews, and after a while, they have reservations booked up for months.

Miquel works six days a week, and that becomes tough. If we compare Miquel to an application architecture, he's a monolithic application that does all the cooking work alone, in sequence. This means the restaurant can only serve a limited number of customers, and they often have to wait a long time to get served.

So, Miquel hires two sous chefs to help him and an assistant who takes over the desserts. Now, when an order comes in, Miquel shouts the order, and the sous chefs and the assistant start preparing their part of the meal. Everyone performs their own task, and together, they create the meal.

As a result, the restaurant serves more clients, and the clients don't have to wait long anymore. This is because Miquel is now a *producer* who sends tasks, the sous chefs and the assistant act as *workers* who listen for new tasks to execute, and Miquel's voice is the *task queue*.

If we look around us, we can see the microservices principles applied everywhere – in our favorite coffee shop, in our team at work, or when we help decorate a friend's new apartment. You always see the three essential components of a *producer*, a *queue*, and a *consumer*.

With this understanding of the microservices architecture and its components, it's time to look at the benefits of microservices so that we can convince stakeholders of why we should adopt microservices.

Listing the benefits of microservices

We have already learned about a significant benefit of microservices – reducing the overall waiting time for the user experience. But luckily, microservices have more benefits that justify our effort to apply them in our applications:

- **Scalability**: Microservices can be scaled up by running as many instances as our hardware or virtual resources allow
- **Flexibility**: Microservices make it easier to adopt new technologies and languages for different parts of your application, enabling flexibility in technology choices
- **Resilience**: A failure in one microservice typically doesn't affect an entire application, enhancing fault tolerance and resilience
- **Maintenance**: Microservices have smaller code bases, which are easier to understand and maintain
- **Self-contained**: Each microservice can run in its own optimized technology stack

- **Loosely coupled**: Microservices are independent of each other and can be built, maintained, and deployed separately
- **Continuous deployment**: Microservices promote **continuous integration and continuous deployment** (**CI/CD**) practices, making it easier to release updates frequently and rapidly

That's quite an impressive list, and it clearly states why microservices are increasingly popular.

The drawbacks of microservices

Looking at the benefits of something always raises the question of whether there are also drawbacks, and, of course, the same goes for microservices. Perhaps we should consider the following points of interest rather than disadvantages. Nevertheless, these are aspects to consider for microservices:

- **Complexity**: Managing microservices can be complex because it requires orchestration, monitoring, and management tools
- **Security**: Each microservice must be secured individually, and task/message queues must also be secured
- **Testing**: Testing a microservices architecture can be more complex, due to the need for integration and end-to-end testing
- **Team coordination**: Effective coordination among multiple teams working on different microservices is crucial

In *Chapter 7*, *Testing Microservices*, and *Chapter 9*, *Securing Microservices*, we'll look at ways to properly deal with the complexity, security, and testing of microservices. For now, we have enough arguments to justify our efforts to apply a microservices architecture. And this allows us to look at two other interesting aspects of microservices – namely, being cloud-native and reactive.

Distinguishing types of microservice

Because microservices are self-contained and independent, they are cloud-native. But what does this mean? And what is cloud-native anyway?

Cloud-native microservices

Let's start with a definition of *cloud-native*. There are many definitions in out there, but they all boil down to this:

Cloud-native is a software architecture for developing and deploying applications in cloud computing environments. With the goal of building scalable, flexible, and resilient applications that fulfill user needs.

As we saw in the *Listing the benefits of microservices* section, microservices are scalable, flexible, and resilient, which makes them very suitable for cloud computing. It also makes them cloud-native by nature.

Let's look at this in the context of commonly used cloud computing environments, such as **Amazon Web Services** (**AWS**), Google Cloud, and Azure Container Apps. To illustrate the possibilities, we will create a microservices architecture that spans several cloud computing platforms:

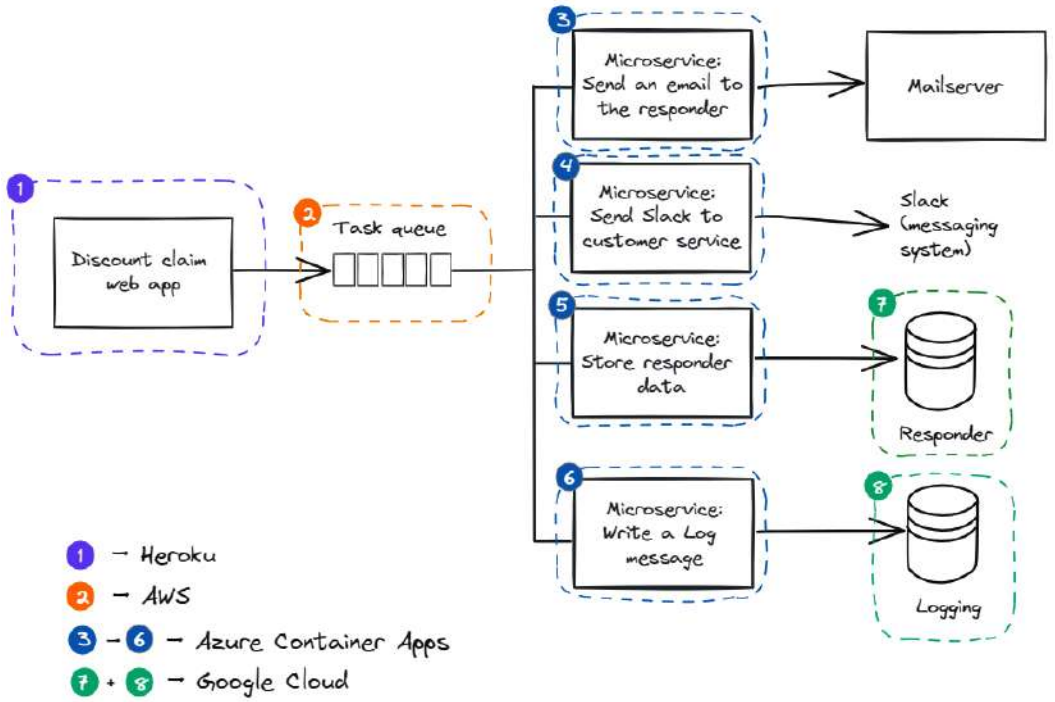

Figure 1.5 – A cloud-native microservices architecture spanning multiple cloud computing environments

This example elaborates on the earlier example in *Figure 1.2* and *Figure 1.4*. The components run as containers on different cloud computing environments as follows:

- (**1**) The discount claim web app runs as a Django app on the Heroku platform.
- (**2**) The task queue is a containerized Celery/RabbitMQ instance that runs on AWS.
- (**3**–**6**) The worker microservices run in separate containers on Azure Container Apps.
- (**7** and **8**) The responder database and the logging database are separate MongoDB databases on Google Cloud.

As indicated, we are exaggerating in this example with four different cloud computing platforms. But if we wanted to, we could construct our microservices application this way. This illustrates the flexible, cloud-native nature of microservices perfectly.

Our web app, the task queue, and the microservices all run in **containers**. Containers are software packages that contain all the necessary parts to run in any environment. They do this by virtualizing the operating system so they can run anywhere, from on-premise to the cloud, or even on a developer's workstation.

This makes containers perfect for running microservices because containers run in any environment, from our laptop to the cloud. Furthermore, containers are self-contained, like microservices, thus strengthening the scalability and resilience of microservices. In *Chapter 8*, *Deploying Microservices with Docker*, we'll explore the details of containers and containerize our microservices.

Reactive microservices

Besides being cloud-native, the microservices we focus on in this book are also reactive. Let's start with a definition again to understand what we mean by reactive:

Reactive microservices are message-driven microservices that run independently and asynchronously while completing a process.

We already saw that our example microservices are message-driven because they listen to a queue for new messages instructing them to execute, which makes them reactive. However, when discussing reactive microservices, you might wonder whether there are also other types of microservices; the answer is yes. For instance, a web API that sends data on request is also a microservice because it is self-contained, independent, decentralized, and specialized.

Therefore, in *Chapter 4*, *Creating RESTful APIs for Microservices*, we also look at building RESTful APIs with Django – not primarily as a microservice in itself but as a service for our reactive microservices. For example, the microservice from *Figure 1.2* that writes a log message could call a RESTful API:

Figure 1.6 – A reactive microservice calling a RESTful API to handle data

This way, we isolate database specifics from our microservice, making it even more resilient and specialized.

Okay, we now know what microservices are, why we should apply them, and what characterizes them. The final part in this chapter focuses on how to design microservices effectively, which we will look at next.

Designing microservices

The general principles of software design also apply to microservices. These principles are out of the scope of this book, and as a developer, you are undoubtedly familiar with them. However, there are a few design aspects that need our special attention when designing microservices:

- **Encapsulating business logic**: Keep the microservice focused on a specific business function and avoid including unrelated functionality.
- **Implementing scalability**: Allow microservices to increase loads by adding more instances. Also, use containerization technologies such as Docker for scalability.
- **Determining the service contract**: Decide on the task or message queue system to use for your microservices.
- **Ensuring loosely coupling**: Minimize dependencies between microservices to allow them to run independently.
- **Choosing the appropriate technology stack**: Select the right programming language and framework to build the microservice based on your requirements.

Let's work out a microservices design to see how we can draft a microservice and address the preceding points. To do so, we will apply the design techniques of *user stories* and *use cases* in the following steps:

1. Analyze a user story that our product owner wrote with the stakeholders.
2. Split the user story into use cases.

We will split the user story into use cases because they have triggers and actors that nicely complement how microservices are set up.

Analyzing the user story

We'll stick with our discount claim example for a while longer and look at the user story that the product owner wrote for this example:

As an early bird responder, I want to claim a discount code for the new AI-powered code editor so that I can buy the code editor at a lower price.

This user story breaks down into three parts:

1. *As an early bird responder* refers to the user who we are developing the feature for.
2. *I want to claim a discount code for the new AI-powered code editor* states the action that the user expects or wants to perform.
3. *So that I can buy the code editor at a lower price* describes why the user uses the feature.

So, the user story stipulates the *who*, the *what*, and the *why*, and this is a good foundation to further design our microservices with use cases.

Split the user story into use cases

The product owner also wrote an explanation of the user story, and after discussing the story, we came up with this list of functions that the application should cover:

- Present a web page to a responder, where they can enter their email address and name to receive a discount code.
- Send an email with the discount code to the responder when they make a claim.
- Send a Slack message to inform customer service about the claim and to follow up.
- Store the responder data in the responder database for later reference.
- Write a log message to the logging database for tracing.

We'll take this list as a starting point, writing the first use case for the web app that presents the web page and acts as a producer to the task-driven worker microservices:

Overview	Present the user with a web page where they can enter their email address and name to receive a discount code.
Actor	Responder/user and a Django web page.
Trigger	The user visits the web page and enter their data.
Steps	The user enters their email address and name. The user clicks the OK button. The web page sends a task to a task queue. The web page informs the user that their request has been received and that an email with a discount code is coming.
Outcome	Feedback to the user that an email with a discount code is coming.

Table 1.1 – The discount claim use case

Next, we'll create the use case that covers sending the email:

Overview	Send an email with a discount code.
Actor	A microservice written in Python.
Trigger	A new task arrives in the task queue.
Steps	Parse the task. Prepare an email. Open a connection to the mail server. Send the email request to the mail server.
Outcome	The responder/user receives the email.

Table 1.2 – The send email use case

This is followed by the use case for sending the Slack message:

Overview	Send a Slack message.
Actor	A microservice written in Python.
Trigger	A new task arrives in the task queue.
Steps	Parse the task. Prepare the message. Open a connection to Slack. Send the message request to Slack.
Outcome	Customer service receives the message.

Table 1.3 – The send Slack message use case

Here is the use case for storing the responder data:

Overview	Store the responder data.
Actor	A microservice written in Python.
Trigger	A new task arrives in the task queue.
Steps	Parse the task. Open a connection to the responder database. Insert the responder data into the database.
Outcome	The database holds the responder data.

Table 1.4 – The store responder data use case

Our design is completed with the use case for writing a log message:

Overview	Write a log message.
Actor	A microservice written in Python.
Trigger	A new task arrives in the task queue.
Steps	Parse the task. Open a connection to the logging database. Insert the log message into the database.
Outcome	The database holds the log message.

Table 1.5 – The write log message use case

Okay, this gives us our use cases as blueprints for developing the microservices. Now, let's check whether we addressed our earlier stated attention points for designing microservices. For our convenience, we will summarize them:

- Encapsulate business logic
- Implement scalability
- Determine the service contract
- Ensure loosely coupling
- Choose the appropriate technology stack

By splitting the user story into use cases with a single task, we *encapsulated business logic* and *ensured loose coupling*. So, that's two ticks on our checklist.

Each use case is a blueprint for a self-contained microservice that is *scalable*. This gives us our third check.

Every microservice use case is triggered by a new task arriving in the task queue. So, we also covered our *service contract*. However, we could take this one step further and also select a specific task queue system, such as RabbitMQ and Celery, which we will cover in detail in *Chapter 6*, *Orchestrating Microservices with Celery and RabbitMQ*.

Finally, we chose Python as the *technology stack* for our microservices, which fulfills our last check.

With that, we have completed our microservices design and also finished learning what microservices are.

Summary

We started this chapter with a definition of microservices. At first, this definition might have sounded a bit abstract. However, by comparing microservices to monolithic applications and looking at the microservices architecture and its components, we provided a clear definition.

Then, we looked at the benefits of microservices that allow us to build more scalable, flexible, and resilient applications. This helped us justify microservices development efforts for our stakeholders. Finally, we explored and applied the software design techniques of user stories and use cases to design microservices.

This prepares the ground for the next chapter, where we will learn about the components of the Django microservices architecture. Also, we'll walk through creating a sample microservice.

2

Introducing the Django Microservices Architecture

Django has its own architecture and components for applying microservices to offload time-consuming tasks. This architecture contains components for building general microservices like RESTful **Application Programming Interfaces** (**APIs**) and reactive microservices like workers who listen to a task queue.

In this chapter, you'll learn about Django's native functionality for building RESTful APIs and caching because APIs improve scalability, and caching optimizes application performance. Furthermore, you'll explore external components like Celery and Redis for reactive microservices that evolved to community standards. And you'll walk through two Django microservices examples so you understand what they consist of and what's involved in building them.

By the end of this chapter, you'll know the Django components for developing microservices. And you understand what makes a Django microservices application and what it takes to build one.

To accomplish this, this chapter deals with the following topics:

- Exploring Django's native components for microservices web applications
- Traversing the external components for Django microservices web applications
- Creating a sample microservice

You now get into our beloved Django territory and are about to discover new and exciting Django components with which you can build sophisticated web applications. Let's dive in.

Technical requirements

The *Creating a sample microservice* section later in this chapter is intended to give you an understanding of what a Django microservice consists of and what it entails to build one. Therefore, you don't need to create them yourself. And you'll set up your development and runtime environment in the next chapter, as you'll be asked to follow along from there.

But you're free to create the samples in this chapter yourself, of course. In that case, you can jump to *Chapter 3*, *Setting Up the Development and Runtime Environment*, to set up your environment and continue here when you are finished.

You can find the code samples for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter02`.

Exploring Django's native components for microservices web applications

Django is a versatile web development tool and has these native components for developing microservices already on board:

- **Django Rest Framework** (**DRF**): A framework for building RESTful APIs as general microservices.
- **Django Cache Framework**: A framework for optimizing the performance of a microservices application.

In the following subsections, we're looking contemplatively at DRF because in *Chapter 5*, *Creating RESTful APIs for Microservices*, we'll explore DRF in depth and build a RESTful API ourselves. The same applies to caching. Later, in *Chapter 10*, *Improving Microservices Performance With Caching*, we'll learn the ins and outs of caching with Django Cache Framework and Redis.

Now we'll address DRF and RESTful APIs.

DRF

We learned in *Chapter 1*, *What Is a Microservice?*, that microservices come in the following basic flavors:

- General microservices like web APIs
- Reactive microservices like a Python worker that listens to a task queue and executes whenever a new task arrives.

Although the focus is on reactive microservices, we also look at general microservices because they're also part of the microservices architecture. And we'll look at one general microservice in particular: a RESTful API and Django's implementation of such an API with DRF.

What is a RESTful API?

We suffice here with a global definition of a RESTful API as we cover this in detail in *Chapter 5*, *Creating RESTful APIs for Microservices*:

A RESTful API is an application programming interface built using the **Representational State Transfer** (**REST**) *architecture*.

The main point of REST is that there is a *request-response*-contract between a producer and a consumer to exchange or retrieve data. Hey, *producer* and *consumer*, that sounds like the components of the microservices architecture, and that's no coincidence, as RESTful APIs are a form of general microservices.

There is one caveat with REST, though, because REST reverses the concepts of producer and consumer: the part that requests data is the *consumer*, and the part that responds with the requested data is the *producer*. So, it is just the opposite of the microservices architecture. Like in the following example, where a Django web application calls a RESTful API to check the combination of a name and bank account:

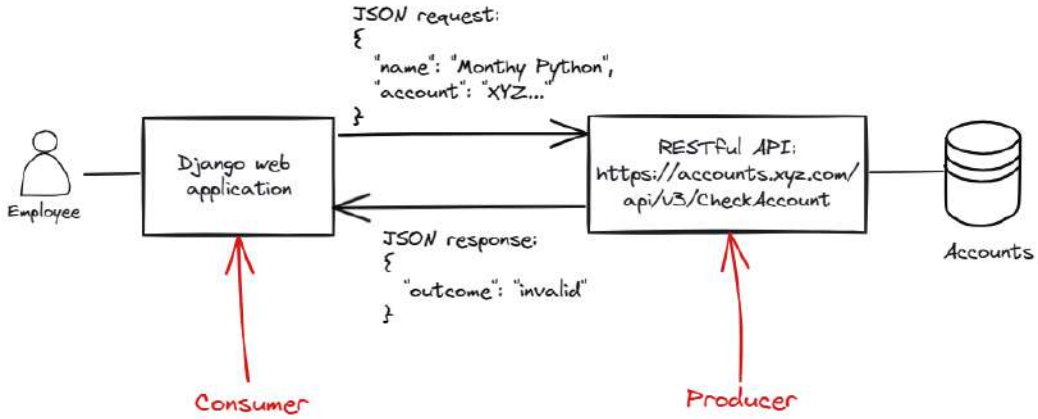

Figure 2.1 – RESTful API with JSON request and response

In this example, the consumer sends a request in JSON format through the `CheckAccount` endpoint to the producer, who reacts by sending back a response in JSON format as stated by the REST contract. An endpoint is a URL that acts as the contact point between a consumer and a producer.

This example also shows a common difference between a general RESTful API microservice and a reactive microservice: a RESTful API microservice sends back data, whereas a reactive microservice, in general, doesn't. Although a reactive microservice can return a response, in most cases, you want to avoid this because it creates dependency.

> **Important note**
>
> If you require a microservice to return a response that the producer must act upon, consider using *Django Channels* and *WebSockets*. Although channels and WebSockets are beyond this book's scope, we'll look at an example later to see how this works.

What is DRF?

DRF is Django's framework for developing RESTful APIs. Even though DRF fully integrates with Django, it is a standalone Python package. Besides handling the HTTP methods `GET`, `POST`, `PUT`, and `DELETE` for data processing, DRF also takes care of *serializing* your data. The DRF serializers convert Django models and querysets into Python data types that you can transform to JSON or XML format.

For example, the RESTful API of *Figure 2.1* could also have a `GetAccount` endpoint that sends back account information. And such a `GetAccount` endpoint could use a Django model (`models.py`) like this:

```
# models.py

from django.db import models

class Account(models.Model):
   name = models.CharField(max_length=50)
   account_number = models.CharField(max_length=34)
```

And a serializer (`serializers.py`) like this:

```
# serializers.py

from rest_framework import serializers
from .models import Account

class AccountSerializer(serializers.ModelSerializer):
   class Meta:
      model = Account
      fields = ('name', 'account_number')
```

And that's all we need, as DRF does the heavy lifting of converting the data for us. Great. In *Chapter 4*, *Cloud-Native Data Processing with MongoDB*, and *Chapter 5*, *Creating RESTful APIs for Microservices*, we go into detail about serializing data. Next, we explore the other native component for Django microservices: Django's Cache Framework.

Django's Cache Framework

The backend most influences the response time of web applications, as it queries databases and renders the requested data. If many users request the same data and the data doesn't change frequently, caching that data can improve application performance dramatically.

A **cache** is a high-speed data storage, generally in the RAM, to store a data subset temporarily to fulfill data requests. Retrieving data from RAM is way faster than retrieving data over a network connection from a database server. So, caching improves application performance significantly.

Let's look at an example of how this works. Say we have an intranet application for internal company news. Every user requests the same latest news items, and the news items rarely change. So, instead of querying the news items database for every user, we can store the latest news items in a cache and serve the users from there. This is much faster because we read from the cache in RAM and prevent the relatively slow database queries.

Because the cache lives in the RAM, it's temporary by nature, and we set a **Time To Live** (**TTL**) to determine how often we'll refresh the cache. Which allows new news items to show up regularly.

The actual caching works according these steps:

1. The first user opens the web page with the latest news items, and the backend retrieves the items from the database, shows them to the user, and places them into the cache.
2. Subsequent users open the web page, and the backend presents them with the items from the cache as long as the TTL validates.
3. When the TTL expires, the cache is refreshed from the database as the first subsequent user requests the page with news items.

Now, let's look at how Django's Cache Framework implements caching.

A Django's Cache Framework example

We can both cache Django apps and Django RESTful APIs. Because this book focuses on microservices, we'll address caching for RESTful APIs because they're also microservices. We'll look at caching in detail in *Chapter 10*, *Improving Microservices Performance With Caching*. For now, we'll finish with an example of applying Django's Cache Framework to our news items example. For completeness, we look at these Django app-level files:

- `models.py`: Contains the definition of the `NewsItem` class.
- `serializers.py`: Converts the `NewItem` querysets into JSON.
- `views.py`: Contains the APIview for news items. This is the file where we'll set the caching.
- `urls.py`: Contains the URL of our API and the connected APIview from the `views.py` file.

Model

We start with defining the `NewsItem` class in our `models.py`:

```
from django.db import models

class NewsItem(models.Model):
   time_stamp = models.DateTimeField(auto_now_add=True)
   title = models.CharField(max_length=150)
   body = models.TextField()
```

```
    def __str__(self):
        return self.title
```

This is just a standard class declaration, as you know from Django. Next, we need to serialize this model to represent it in JSON format.

Serializer

The `serializers.py` file specifies which model fields we want to serialize:

```
from rest_framework import serializers as SL
from .models import NewsItem

class NewsItemSerializer(SL.HyperlinkedModelSerializer):
    class Meta:
        model = NewsItem
        fields = ('time_stamp', 'title', 'body')
```

We serialize all fields from the model and are ready to set up the view.

Important note

We can choose between the `HyperlinkedModelSerializer` and the `ModelSerializer` to serialize fields. The main difference is that the `HyperlinkedModelSerializer` response includes complete URLs and `ModelSerializer` only key values.

`HyperlinkedModelSerializer`:

```
  ' newsitems ':
  [
    'http:127.0.0.1:8000/newitems/1',
    'http:127.0.0.1:8000/newitems/2'
  ]
```

`ModelSerializer`:

```
  ' newsitems ': [
    1,
    2
  ]
```

View

Next, we'll specify the APIview for retrieving news items with the HTTP `GET` method in `views.py`:

```
 1 from django.utils.decorators import method_decorator
 2 from django.views.decorators.cache import cache_page
 3 from rest_framework import status
 4 from rest_framework.views import APIView
 5 from rest_framework import permissions
 6 from rest_framework.response import Response
 7 from .serializers import NewsItemSerializer
 8 from .models import NewsItem
 9
10 class NewsItemApiView(APIView):
11    permission_classes = [permissions.IsAuthenticated]
12
13    @method_decorator(cache_page(15 * 60))
14    def get(self, request, *args, **kwargs):
15       new_items = NewsItem.objects.all().
                     order_by('time_stamp')
16       serializer = NewsItemSerializer(new_items,
                                         many=True)
17       return Response(serializer.data,
                         status=status.HTTP_200_OK)
```

This is where the caching magic takes place. *Lines 1* and *2* import the caching methods from Django's Cache Framework.

Line 13 decorates the `get` function so its result gets cached when handling a request. The `@method_decorator` uses the `cache_page` method to set the TTL to 15 minutes (15 times 60 seconds). Together, they steer the caching of news items. We leave the rest of `views.py` for now because we learn more about the views of RESTful APIs in *Chapter 5*, *Creating RESTful APIs for Microservices*.

URL

The example is almost done. We only need to make the APIview of news items accessible to the outer world through `urls.py`:

```
from django.urls import include, path
from .views import NewsItemApiView

urlpatterns = [
    path('api/news', NewsItemApiView.as_view())
]
```

And that's it. Caching now optimizes the application's performance. Applying Django's Cache Framework only takes three lines (*lines 1*, *2*, and *13* in `views.py`) of extra code on top of standard Django code. Quite impressive and highly effective.

Okay, for now, we have a good impression of Django's native components for microservices. So, let's complete our overview of the Django microservices architecture and look at external components like Celery.

Traversing the external components for Django microservices web applications

In *Chapter 1*, *What Is a Microservice* we've addressed these main components of a microservices architecture:

- Producer
- Task queue
- Worker

In the previous section, we saw that Django apps can act as producers and we can build reactive microservices (workers) in Python But for a complete microservices architecture, we also need a task queue. Furthermore, it's desirable to have container software to facilitate the scalability and independency of the components.

Django doesn't have task queues and containerization on board, so this is where the external components for Django microservices come into play. First, let's look into task and message queue brokers.

Task and message queue brokers

For task and message queueing, Django collaborates with the following external components:

Component	**Purpose**	**Type**
Celery	Task queue broker	Python package
RabbitMQ	Message queue broker	Stand-alone software
Redis	Message queue broker and caching system	Stand-alone software

Table 2.1 – Task and message queue brokers for Django

There are other task and message queueing brokers, such as Dramatiq. But Celery, RabbitMQ, and Redis are the most common in the Django community; therefore, we will focus on Celery, RabbitMQ and Redis.

We'll start with Celery.

Celery

Celery is an open-source task queue system written in Python and available as a Python package (`https://pypi.org/project/celery/`). Celery offers this functionality:

- Offload tasks asynchronously
- Schedule tasks

Because we focus on reactive microservices, we'll only address offloading tasks with Celery.

Celery runs on top of message queue brokers like RabbitMQ and Redis. Celery has become a standard for Django microservices because Celery:

- Abstracts the technical details of RabbitMQ and Redis and lets us concentrate on offloading tasks from Django.
- Integrates well with Django.

Chapter 6, *Orchestrating Microservices With Celery and RabbitMQ*, will teach us how to orchestrate microservices with Celery and RabbitMQ. And in *Chapter 10*, *Improving Microservices Performance With Caching*, we'll use Redis. Because of this, you'll see all three in action, which helps you choose which is best for your situation.

> **Important note**
>
> Celery only works on UNIX-based operating systems like Linux and Mac OS. If you're on Windows 10 or 11, consider using the **Windows Subsystem for Linux** (**WSL**). WSL allows you to run an *Ubuntu* instance where you can use Celery.
>
> In *Chapter 3*, *Setting Up the Development and Runtime Environment*, we'll set this up for Windows, including the WSL extension for **Visual Studio Code** (**VS Code**), which allows us to develop applications on the Ubuntu instance.

You can find more information about Celery at `https://docs.celeryq.dev/en/stable/`.

RabbitMQ

RabbitMQ is an open-source message queue broker based on the **Advanced Message Queuing Protocol** (**AMQP**). We can install and run it on a server, but the easiest way is to run it in a Docker container on top of the standard Docker image for RabbitMQ. Either way, RabbitMQ is available then on TCP port 15672. In *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ* , we'll see how we apply the RabbitMQ Docker image and address its TCP port.

You can find more information about RabbitMQ at `https://www.rabbitmq.com/documentation.html`.

Redis

Redis is an open-source, in-memory data store that provides the following from a Django perspective:

- Message queue brokering
- Caching in collaboration with Django's Cache Framework
- In memory database functionality

Redis uses the **REdis Serialization Protocol** (**RESP**). Like RabbitMQ, we can run Redis on a server or Docker. Its standard TCP port is 6379. And Redis can also run as a vehicle for Celery.

Furthermore, Redis can handle caching. Later, in *Chapter 10*, *Improving Microservices Performance With Caching*, we'll use Redis for both message brokering and caching.

You can find more information about Redis at `https://developer.redis.com/`.

Container software

Container software isn't part of the Django microservices architecture in the strict sense. However, since containerizing microservices facilitates scaling and independence, we still consider it an external component for microservices.

Despite there being multiple containerization systems, we'll only look at Docker because it's the most used option.

Docker

Docker is an open-source tool for containerizing applications with these main parts:

- **Docker Containers** that encapsulate an application and its dependencies.
- **Docker Images** that act as blueprints for the containers.
- **Docker Engine** that runs the containers.

Containers are software packages that contain all parts to run in any environment. They can run anywhere because they virtualize the operating system. Docker builds containers from images, which are software package blueprints—for example, a RabbitMQ container derived from the RabbitMQ image. And the Docker Engine runs the containers and includes a **Command Line Interface** (**CLI**) to interact with Docker.

Regarding Django microservices, our Djano app, the task/message queue broker, and the microservices can all run as separate Docker containers.

You can find more information about Docker at `https://docs.docker.com/`.

The complete Django microservices architecture

In *Part 2*, *Building the Microservices Foundation*, and *Part 3*, *Taking Microservices to Production Level*, we cover the components of Django's microservice architecture in detail. Therefore, we suffice here with an overview of a Django microservices architecture where Celery and Redis orchestrate the tasks between producers and workers, and Docker containerizes the components:

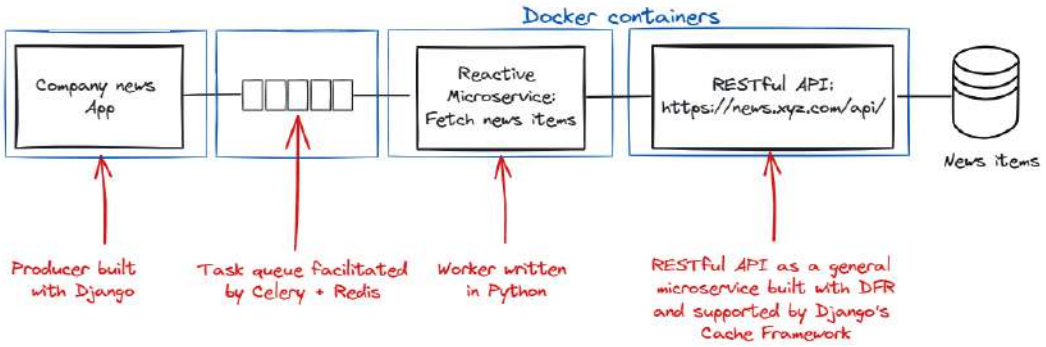

Figure 2.2 – Django microservices architecture with DRF, Django's Cache Framework, and Celery/Redis

With this overview, we have all we need to compose Django Microservices—it's time to take the next step and explore creating a sample microservice.

Creating a sample microservice

Okay, until now, we focused on what microservices are and what the Django microservices architecture consists of. With what we've gathered so far, we're ready to look at a complete microservice to understand what it consists of and what it takes to build one. We'll split this up into two implementations. First, we'll explore a microservice implementation with RabbitMQ to understand how message queueing technically works. Then, we'll walk through a Django microservice implementation with Celery to see what a task-driven Django microservice comprises.

Important note

The Django developer community often uses Celery for task queue management because it's simple and effective since Celery abstracts the technical details of the underlying message queuing broker.

Nevertheless, it's valuable to know the inners of message queueing with RabbitMQ and Redis as well because they serve as a vehicle for Celery, and you might encounter situations where you prefer using RabbitMQ or Redis directly.

So, we'll look into developing with RabbitMQ, Redis, and Celery. That way, you'll be able to apply them as needed.

The implementations cover the same functionality, sending an email from a microservice, giving us good insight into the differences between a RabbitMQ microservice and a Celery microservice, which we'll cover next.

Our sample Django app asks visitors to send a suggestion for improvement. It opens with this form:

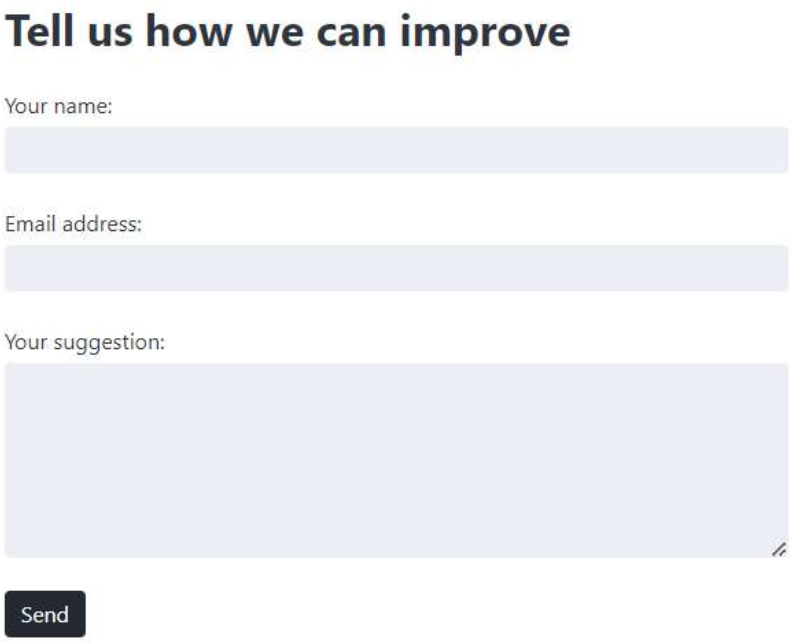

Figure 2.3 – The Improvement Suggestion app

After filling out the form and clicking **Send**, the app offloads a task to send a confirmation email to the user. This way, the user experience remains smooth, even if sending the mail takes 10 seconds.

In the next section, we'll go through a RabbitMQ implementation for the email microservice.

Implementation 1: Offloading a task with a RabbitMQ microservice

RabbitMQ has several methods for sending a message from a producer to a consumer/worker. *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ*, discusses these methods in more detail. Here, we suffice with the *work queue* method where RabbitMQ distributes the tasks among multiple workers.

You can find the code for this implementation on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter02/impl1`.

In the upcoming subsections, we'll address the following:

- The directory structure and settings for the Django project.
- The producer file, which off loads the tasks for sending mails.
- The worker file, which sends the confirmation email.
- The form and the view for the Django app.

Directory structure

This is the main directory structure for the Django project with the relevant files:

```
impl1/
|
├── django_rabbitmq/
|   └── settings.py
|
├── scripts/
|   ├── __init__.py
|   └── worker.py
|
├── suggestion/
|   ├── templates/suggestion
|   |   └── suggestion.html
|   |
|   ├── forms.py
|   ├── producer.py
|   └── views.py
|
└── manage.py
```

The structure only shows the files which are relevant for the microservice handling. Also, the `venv`-directory is left out but required, of course.

Django-extensions and the scripts directory

We'll get to the `django_rabbitmq` directory in a moment, but first, we'll look at the `scripts` directory. To run the consumer/worker that sends the email within the Django context, we enable it as a Django extension via the `django-extensions` package. This package assumes a `scripts` directory, including an empty `__init__.py` file, where we install extensions like our email worker.

Settings

This also brings us to `settings.py`, where the following stands out:

```
INSTALLED_APPS = [
    ...
    "django_extensions",
    "suggestion.apps.SuggestionConfig",
]

EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

The `django-extensions` package must be defined as an installed app, like the `suggestion` app. We'll set the console as the email backend to quickly check sent emails without the hustle of setting up a mail server or using external mail servers.

The form and the producer

The suggestion form from *Figure 2.3* derives from this set up for the `forms.py` file:

```
 1 from django import forms
 2 from suggestion.producer import send_email_task_message
 3
 4 class SuggestionForm(forms.Form):
 5     name = forms.CharField(label="Your name")
 6     email = forms.EmailField(label="Email")
 7     suggestion = forms.CharField(
             label="Your suggestion",
             widget=forms.Textarea(attrs={"rows": 7})
       )
 8
 9     def send_email(self):
10         task_message = {
              "name": self.cleaned_data["name"],
              "email": self.cleaned_data["email"],
              "suggestion": self.cleaned_data["suggestion"]
           }
11         send_email_task_message(task_message)
```

Line 2 imports the producer to enable sending a task message to a consumer/worker.

Lines 9-11 extend the straightforward class definition for the suggestion form with the `send_email` method.

Line 10 creates the task message in JSON format to send to the `message_queue`, and the task message contains parameters for the task to process.

In the end, *line 11* calls the `send_email_task_message` function in the `producer.py` file, which looks like this:

```
 1 import json
 2 import pika
 3
 4 connection =
 5 pika.BlockingConnection(pika.ConnectionParameters(
 6 host='localhost',
 7 credentials=pika.PlainCredentials("myuser",
                                    "mypassword")))
 8
 9 channel = connection.channel()
10 channel.queue_declare(queue='mail_queue',
                        durable=True)
11
12 def send_email_task_message(task_message):
13    channel.basic_publish(exchange= "",
            routing_key='mail_queue',
            body=json.dumps(task_message),
            properties=pika.BasicProperties(
            delivery_mode=
              pika.spec.PERSISTENT_DELIVERY_MODE)
      )
```

The microservices architecture starts here by utilizing the producer, which opens the task message queue. To do so, *line 2* imports the `pika` package, which implements the AMQP protocol communicating with RabbitMQ.

Lines 4-7 open the connection with RabbitMQ. The credentials from *line 7* are for demoing only. Next, *lines 9-10* create a task message queue named `mail_queue`.

Line 12 sets the earlier mentioned `send_email_taskmessage` function.

Line 13 sends a task message to the `mail_queue`.

Template

The accompanying form template, `suggestion.html`, is straightforward:

```
{% extends 'suggestion/base.html' %}

{% block content %}

<h1>Tell us how we can improve</h1>

<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Send" />
</form>

{% endblock content %}
```

This template extends the base.html template and formats the form as HTML paragraphs.

The view

From Django's perspective, this `views.py` file steers the form:

```
from django.views.generic.base import TemplateView
from django.views.generic.edit import FormView
from suggestion.forms import SuggestionForm

class SuggestionFormView(FormView):
    template_name = "suggestion/suggestion.html"
    form_class = SuggestionForm
    success_url = "/success/"

    def form_valid(self, form):
        form.send_email()
        return super().form_valid(form)

class SuccessView(TemplateView):
    template_name = "suggestion/success.html"
```

Here, the `form_valid` method calls the `send_email` method, which sets off the process of sending a task message to the task message queue.

The email consumer/worker

Finally, the consumer/worker in `worker.py` listens to the `mail_queue` and sends an email when a new task message arrives:

```
 1 import json
 2 import pika
 3 from django.core.mail import send_mail
 4 from time import sleep
 5
 6 connection =
 7 pika.BlockingConnection(pika.ConnectionParameters(
 8     host='localhost',
 9     credentials=pika.PlainCredentials("myuser",
                                         "mypassword")))
10
11 channel = connection.channel()
12 channel.queue_declare(queue='mail_queue',
                         durable=True)
13
14 def callback(ch, method, properties, body):
15     task_message = json.loads(body)
16     sleep(10)
17     send_mail(
18         "Your suggestion for improving",
           f"We'll include your suggestion -
            {task_message['suggestion']} -
            into our improvement process.\n\n
            Thanks for your contribution!",
           "quality@xyz.com",
           [task_message['email']],
           fail_silently=False,
       )
19     ch.basic_ack(delivery_tag=method.delivery_tag)
20
21 def run():
22     channel.basic_qos(prefetch_count=1)
23     channel.basic_consume(queue='mail_queue',
                             on_message_callback=callback)
24     channel.start_consuming()
```

Line 3 imports the Django `send_mail` method.

Lines 6-12 connect the worker to the `mail_queue`.

Lines 14-19 set the `callback` function, which runs every time a new task message arrives. This function parses the received task message and emulates a delay in *line 16* to emphasize the asynchronous processing of the worker.

Line 21 sets the `run` function, which enables the worker to act as a Django extension and run inside Django's context.

Finally, *lines 22-24* cause the worker to listen to the `mail_queue` and call the `callback` function when a new task message arrives.

RabbitMQ as message queue broker

Besides the producer and consumer/worker, this implementation requires a RabbitMQ instance, which handles the queues and messages. You can run RabbitMQ by downloading and installing it for the RabbitMQ website (`https://www.rabbitmq.com/download.html`). Or you can run RabbitMQ from a Docker container, as *Chapter 8*, *Deploying Microservices With Docker*, explains. Here, we assume RagbitMQ runs in a container, though it makes no difference for the Django code.

Putting it to work

Okay, we have everything prepared. We activate our virtual environment from a terminal shell and launch the Improvement Suggestion app, including the producer:

```
$ python3 manage.py runserver
```

And in a second terminal, we launch the worker as a Django extension:

```
$ python3 manage.py runscript worker
```

This sets the RabbitMQ worker to wait for new messages:

```
Waiting for email requests
```

Figure 2.4 – Worker waiting prompt

If we now open a browser, navigate to `http://127.0.0.1:8000/suggestion`, and fill out the form, the app executes these steps:

1. The `form_valid` method in `views.py` fires the `send_email` method from `forms.py`.
2. The app presents the success form.
3. The `send_email` method in `forms.py` composes the task message, calls the `send_email_task_message` function in `producer.py`, and passes the message.
4. The `send_email_task_message` function in `producer.py` sends the passed task message to the `mail_queue`.

5. The consumer/worker detects a new task message and then launches the `callback` function to parse the task message and send the confirmation email. After the simulated delay, the worker terminal shows the email:

```
Subject: Your suggestion for improving
From: quality@xyz.com
To: woldman@tw-techwriter.nl
Date: Wed, 15 Nov 2023 07:36:03 -0000
Message-ID:
 <170003376353.1442.12929963173750605390@DESKTOP-V3R0DRC.localdomain>

We'll include your suggestion - Deliveries on Saturdays. - into our improvement process.

Thanks for your contribution!
```

Figure 2.5 – Email to console (RabbitMQ)

There you have it, the first implementation of a Django microservice.

Implementation 2: Offloading a task with a Celery microservice

Next, we'll walk through the Celery implementation for the email microservice. We'll only look at the parts that differ from the RabbitMQ implementation.

You can find the code for this implementation on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter02/impl2`.

In the following subsections, we'll look into the Celery settings, the forms, and the Celery tasks file.

Directory structure

The directory structure is slightly different for Celery:

```
impl2/
|
├── django_celery/
|   ├── __init__.py
|   ├── celery.py
|   └── settings.py
|
├── suggestion/
|   ├── forms.py
|   └── tasks.py
|
└── manage.py
```

Again, we leave out the standard Django files.

Defining Celery as an app

Celery makes itself known as an app to Django through this `celery.py` file:

```
import os
from celery import Celery

os.environ.setdefault("DJANGO_SETTINGS_MODULE",
                      "django_celery.settings")
app = Celery("django_celery")
app.config_from_object("django.conf:settings",
                       namespace="CELERY")
app.autodiscover_tasks()
```

And by exposing the Celery app via the `__init__.py` file in the `django_celery` directory:

```
from .celery import app as celery_app
__all__ = ("celery_app",)
```

This sets the `django_celery` directory as a module, which we can use to create a Celery worker.

Settings

We cloud use RabbitMQ as the vehicle for Celery, but to show Celery's versatility, we use Redis here as a message broker through these settings:

```
CELERY_BROKER_URL = "redis://localhost:6379"
CELERY_RESULT_BACKEND = "redis://localhost:6379"
```

Both settings refer to the same URL, but we could also have the broker refer to RabbitMQ and the result backend to Redis.

The form and the producer

Celery enables us to combine the producer directly into the form through `forms.py`:

```
1 from django import forms
2 from suggestion.tasks import send_email_task
3
4 class SuggestionForm(forms.Form):
5     name = forms.CharField(label="Your name")
6     email = forms.EmailField(label="Email address")
7     suggestion = forms.CharField(
8         label="Your suggestion",
```

```
 9          widget=forms.Textarea(attrs={"rows": 7})
10      )
11
12      def send_email(self):
13          send_email_task.delay(
               self.cleaned_data["name"],
               self.cleaned_data["email"],
               self.cleaned_data["suggestion"]
            )
```

Line 2 imports the producer task `send_email_task` from `tasks.py`. In Celery, the producer and the consumer/worker refer to the *same task functions*: the producer calls the task function, and the worker listens to task function calls.

So, *line 12* sets the `send_email` function, which calls the shared `send_email_task` function in `tasks.py` with the Celery `delay` method to send the task to the queue.

The worker

Celery expects the worker tasks in the `tasks.py` file:

```
1 from time import sleep
2 from django.core.mail import send_mail
3 from celery import shared_task
4
5 @shared_task()
6 def send_email_task(name, email_address, suggestion):
7     sleep(10)
8     send_mail(
        "Your suggestion",
        f"We'll include your suggestion - {suggestion} -
         into our improvement process.\n\nThanks for
           your contribution!",
       "quality@xyz.com",
       [email_address],
       fail_silently=False,
    )
```

Line 3 imports the `shared_task` decorator as applied in *line 5*. It ensures the worker notices every call from a producer to the `send_email_task` function and then processes all the functions steps. And this is all that's needed to define a function as an asynchronous task. Quite impressive.

This clearly illustrates how Celery abstracts the details of the underlying message queue. In fact, it doesn't open a named queue and leaves that to Redis. This way, Celery's implementation is easier and more compact than the RabbitMQ implementation, which explains Celery's popularity.

Redis serving as the message broker for Celery

We assume Redis runs in a Docker container, as we'll see in *Chapter 8*, *Deploying Microservices With Docker*.

Putting it to work

This implementation requires the `celery` Python package, and once installed, we launch the worker from a terminal:

```
$ celery -A django_celery worker -l info
```

We go into the details of this command in *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ*. Here, the command sets the Celery worker to wait for new messages, and the terminal shows that Redis takes care of transporting the messages:

```
 -------------- celery@DESKTOP-V3R0DRC v5.3.5 (emerald-rush)
--- ***** -----
-- ******* ---- Linux-5.10.102.1-microsoft-standard-WSL2-x86_64-with-glibc2.35 2023-11-15 08:01:34
- *** --- * ---
- ** ---------- [config]
- ** ---------- .> app:         django_celery:0x7f55aaa12010
- ** ---------- .> transport:   redis://localhost:6379//
- ** ---------- .> results:     redis://localhost:6379/
- *** --- * --- .> concurrency: 16 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery
```

Figure 2.6 – Celery status information

Next, we start the Django app, navigate to `http://127.0.0.1:8000/suggestion`, and fill out the form, which fires these steps:

1. The `form_valid` method in `views.py` fires the `send_email` method from `forms.py`.
2. The app shows the success form.
3. The `send_email` method in `forms.py` sends a task to the task queue through `send_email_task.delay`.
4. The worker detects the call for the `send_email_task` function and executes the function's steps to send the confirmation email. After the simulated delay, the Celery worker terminal shows the email:

```
Subject: Your suggestion
From: quality@xyz.com
To: woldman@tw-techwriter.nl
Date: Wed, 15 Nov 2023 08:12:15 -0000
Message-ID:
 <170003593571.7132.15422821944286215066@DESKTOP-V3R0DRC.localdomain>

We'll include your suggestion - Just keep up the good work. - into our improvement process.

Thanks for your contribution!
```

Figure 2.7 – Email to console (Celery)

And that completes the Celery implementation of the Django microservice. And also completes the introduction to the Django microservices architecture.

Summary

This chapter introduced you to the Django microservices architecture. First, we explored Django's native components within the microservices architecture. Next, we completed the architecture with external microservices components such as Celery for task queue management. Finally, we went through the implementations of a RabbitMQ microservice and a Celery microservice.

With this, you understand what makes a Django microservices application and what it takes to build one. Furthermore, you know the differences between a RabbitMQ and a Celery microservice, which helps you decide your best solution.

In the next chapter, we'll prepare our development and runtime environment, enabling us to develop microservices.

3

Setting Up the Development and Runtime Environment

We've finished the introduction of Django microservices and are now ready to set up our development and runtime environment so we can start building microservices.

In this chapter, you'll learn what applications and Python packages are needed to build Django microservices and how you install them. Furthermore, you'll be introduced to the sample microservices application you will develop throughout the following chapters.

By the end of this chapter, you'll know how to set up your toolset for developing Django microservices, and you'll be ready to start building. In addition, you'll have a clear picture of the microservices application that you'll build step by step over the course of this book.

Hence, this chapter addresses the following topics:

- Setting up the development environment
- Setting up the runtime environment
- Analyzing the sample microservices application

With this, we'll have everything in place to start developing, and that's, in the end, what we're all about. So get ready and follow along!

Technical requirements

This chapter's remainder describes the applications and Python packages needed to develop and run Django microservices. As a start, you need a workstation running Linux, Mac OS, or Windows 10/11.

Since Celery doesn't run on Windows, you may wonder why Windows 10/11 is still an option. This is because **Windows Subsystem for Linux** (**WSL**) allows you to run virtual Linux distributions on Windows 10/11, such as the Ubuntu distribution.

Furthermore, we'll use **Visual Studio Code** (**VS Code**) as our code editor because it's available for Linux, Mac OS, and Windows. And because it integrates well with WSL and MongoDB, the database that we'll be using.

Altogether, you'll need the following:

- A (virtual) workstation running one of these operating systems:
  - Linux, such as Ubuntu 22.04 LTS
  - Mac OS
  - Windows 10 version 2004 or newer
  - Windows 11
- Python version 3.8 or newer
- VS Code

This chapter doesn't contain code examples, but you're required to install a list of Python packages, which are gathered in a `requirements.txt` file at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter03`.

Setting up the development environment

As developers, we need a toolbook to work with and get started. So, we'll begin with our development environment. Because Linux also runs on Windows through WSL and Linux shell commands overlap with Mac OS commands, this book utilizes Linux as a development and runtime platform. So we catch three birds with one stone.

If you're on Windows, you first need to install WSL and Ubuntu as described in the subsection, *Extra setup for Windows developers*. If you're on Linux or Mac OS, you can skip this subsection and jump to the *Installing the required Python packages* subsection.

Extra setup for Windows developers

If you're on Windows 10 or 11, you must activate WSL and install an Ubuntu instance to follow along. You also need to install Windows Terminal and integrate WSL with VS Code.

Setting up WSL and Ubuntu is easy:

1. Open your BIOS settings and check if virtualization is enabled. If needed, enable virtualization.
2. Start or restart your workstation.
3. Open PowerShell in administrator mode.

4. Enter this command to install WSL:

```
$ wsl --install
```

5. Follow the instructions and restart your workstation when requested.

These steps install WSL and the Ubuntu Linux distribution. To check your WSL and Ubuntu version, open PowerShell again and type this command:

```
$ wsl -l -v
```

You should see that Ubuntu runs under WSL version 2.

For more information on WSL, check the Microsoft Learn site: `https://learn.microsoft.com/en-us/windows/wsl/`.

Installing Windows Terminal

Furthermore, you'll need the **Windows Terminal application**, which you can install from Microsoft Store: `https://apps.microsoft.com/detail/windows-terminal/9N0DX20HK701?hl=en-US&gl=US`.

Once you've installed WSL and Windows Terminal, start Windows Terminal and click the downwards arrow in the menu bar, which opens this menu:

Figure 3.1 – Menu options for Windows Terminal

Select **Ubuntu** to open an Ubuntu shell to check if the installation works. You should see a shell like this:

Figure 3.2 – Ubuntu shell

If the shell opens, we know Ubuntu works, but to be sure we can check its version with this command:

```
$ lsb_release -a
```

You should see the version in the `Release` item in the command output.

Python is already on board

Ubuntu has Python already installed, and you can check the Python version through the shell:

```
$ python3 --version
```

Most likely, WSL installed Ubuntu version 20.04 or 22.04. Version 20.04 comes with Python 3.8, and version 22.04 with Python 3.10. To ensure Python works correctly, you need to execute the following commands:

```
$ sudo apt update && sudo apt upgrade
$ sudo apt upgrade python3
$ sudo apt install python3-pip
$ sudo apt install python3-venv
```

When finished, you're ready to set up your development environment with VS Code.

Integrating VS Code and WSL

VS Code integrates well with WSL. All you need to do is install the **WSL extension** in VS Code:

1. Open VS Code.
2. Click the **Extensions** button at the left menu or type *Ctrl* + *Shift* + *X* to open the **Extensions** panel.
3. In the **Search** field, enter *WSL*.

   * *VS Code shows a list of matching extensions.*
4. Click the WSL extension from the list.

   * *A central panel with WSL information opens.*
5. Click **Install** in the main panel to add the extension.
6. Close VS Code.

VS Code can now access your WSL Ubuntu version. Take the following steps to see how this works:

1. Open an Ubuntu shell in Windows Terminal.
2. Type this command to open your current directory in VS Code:

```
$ code .
```

This opens VS Code, showing the current directory and its content. In the left lower corner of VS Code, you'll see *WSL: Ubuntu*, which indicates you're running VS Code on Ubuntu.

Okay, you're all set now and can continue installing Python packages.

Installing the required Python packages

Well, now it's Python's turn. If you're on Mac OS, you might need to install Python first, but since this book assumes you've Django experience, you almost certainly already have Python in place. If not, download Python from `https://www.python.org/downloads/macos/` and install it on your workstation.

To install the required Python packages, we first create the Django projects directory for the sample application where we can install the required Python packages.

> **Important note**
>
> The shell commands in this chapter and throughout the book are probably trivial to Linux and Mac OS developers. But they're likely new to Windows developers, so we explicitly describe them.

First, we open a terminal shell and create the `projects` directory:

```
$ mkdir django-microservices
```

And move into this new directory:

```
$ cd django-microservices
```

> **Important note**
>
> From here, we'll assume a terminal shell as a command entry point and won't repeat the instructions to open a terminal again.

Now create a Python virtual environment:

```
$ python3 -m venv venv --prompt="micro"
```

The `--prompt` option isn't required, but it gives us a nice indicator of the virtual environment we are working in after we activate the environment:

```
$ source venv/bin/activate
```

As you see, the indicator `micro` precedes your shell prompt. Excellent, you're set now to install these required Python packages:

- `django`: although this goes without saying.
- `django-extensions`: to run independent workers inside Django's context.
- `djangorestframework`: to build **REpresentational State Transfer Application Programming Interfaces** (**RESTful APIs**) with Django.
- `requests`: to interact with RESTful APIs.
- `pymongo`: to connect to and work with a MongoDB database as our application will do.
- `djongo`: to enable ORM for MongoDB.
- `pytz`: required for `djongo`.
- `pika`: to use a RabbitMQ message broker directly to send and receive task messages.
- `celery`: to offload tasks to workers from Django applications.
- `flower`: to monitor Celery task execution.
- `redis`: to enable Redis to serve as a vehicle for Celery.

We can install these packages at once by downloading the `requirements.txt` file at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter03` into the projects directory and inputting this file to `pip`, the Python package installer:

```
$ pip install -r requirements.txt
```

> **Important note**
>
> If you're getting restrictions or access errors, change the access rights for the `venv` directory with this command:
>
> **`$ sudo chmod -R a+rwx venv`**
>
> And try to install the packages again.

If you want complete control of the installation or want to see how installing packages individually works, you can install each package one by one. For example, you install Django this way:

```
$ pip install Django
```

You install the other packages the same way with the remark that you need specific versions of `djongo` and `pymongo`:

```
$ pip install djongo==1.3.6
$ pip install pymongo==3.12.3
```

When you're finished, you can check the installed packages with this command:

```
$ pip list
```

The displayed list of packages should contain the content of the `requirements.txt` file, and you'll see a number of dependent packages that pip installed automatically. For instance, the `celery` package requires the `kombu` package. And the `kombu` package, in turn, depends on the `amqp` package. You can check such dependencies with this command:

```
$ pip show <package-name>
```

Great, you've set up your development environment. Now, let's finish our setup with the runtime environment.

Setting up the runtime environment

For our runtime environment, we'll set up the following software:

- **RabbitMQ**: to act as a message broker.
- **Redis**: to act as a message-handling vehicle for Celery and to provide caching.
- **MongoDB**: to store our data.
- **Docker**: to containerize our software and microservices.

MongoDB is the database that our microservices application will use. We cloud use PostgreSQL, MySQL, or another database as well, but MongoDB can run entirely in the cloud, which aligns nicely with our principle of building cloud-native microservices.

Because we'll run RabbitMQ and Redis as Docker containers, we'll start with installing Docker Desktop.

Installing Docker Desktop

Docker Desktop allows you to containerize applications on your workstation, and the installation depends on your platform:

Platform	Instructions/URL
Linux distributions Windows developers need to follow this route as well for WSL	Follow the instructions for installing and running Docker Desktop at `https://docs.docker.com/desktop/install/linux-install/`.
Mac OS	Download your Intel or Apple chip version from `https://www.docker.com/products/docker-desktop/` and install it on your workstation.

Table 3.1 – Installation routes for Docker Desktop

After installing and starting Docker Desktop, check if Docker runs:

```
$ docker –version
```

The terminal output should show your Docker version, indicating that Docker is ready to serve and run.

Configuring a Docker network bridge to connect containers

The microservices application we'll build will be a so-called *multi-container* application because it spans several containers. For example, we'll have a Django app and a microservice worker running in separate containers. Docker facilitates such multi-container applications (and the communication between the respective containers) in two ways:

1. By connecting multiple independent containers through a *bridge network*.
2. Through the *Docker Compose* tool, which creates and starts multiple containers at once within the same network context.

We'll cover both ways and prepare a bridge network here. We name the network `my-network`, but can give any name we like:

```
$ docker network create -d bridge my-network
```

In the next subsection, we'll add a RabbitMQ container to this network; in the ensuing subsection, we'll add a Redis container.

Okay, we've Docker installed and prepared. In *Chapter 8*, *Deploy Microservices with Docker*, we'll further explore Docker, including creating a multi-container application with Docker Compose.

Installing RabbitMQ as a Docker container

We can install RabbitMQ as a server application on our workstation and as a Docker container. Since we focus on containerizing microservices, we install `rabbitMQ` as a Docker container and add it to the bridge network with this command:

```
$ docker run -it --rm --name my-rabbitmq
  --network my-network -p 5672:5672 -p 15672:15672
  rabbitmq:3.8-management-alpine
```

The `--name` option sets the container name. The `--network` option connects the containers to our Docker bridge network, and the `-p` option specifies the **Transmission Control Protocol** (**TCP**) ports. Finally, `rabbitmq:3.8-management-alpine` sets the RabbitMQ image that acts as a blueprint for the container.

We can check through Docker Desktop if the container runs with this command:

```
$ docker ps -a
```

The command result should show an item for your RabbitMQ container, which starts with the `CONTAINER ID`.

To ensure that we correctly added the RabbitMQ container to our bridge network, enter this command:

```
$ docker network inspect my-network
```

In the command output, you should see an entry like this for the RabbitMQ container:

```
"Containers": {
   "c789042f5...": {
      "Name": "my-rabbitmq",
      "EndpointID": "c3e9c27148d...",
      "MacAddress": "00:00:xy:00:00:00",
      "IPv4Address": "100.23.0.0/10",
      "IPv6Address": ""
   }
},
```

With this, RabbitMQ is ready process task messages.

Installing Redis as a Docker container

We also install Redis as a Docker container and add it to the network with this command:

```
$ docker run --name my-redis --network my-network -d -p 6379:6379
redis
```

Again, the `--name` option and the `--network` option set the container name and add the container to the bridge network. Furthermore, we configure the TCP port and base our Redis container on the `redis` image.

Now, check for yourself if the Redis container is running and correctly added to the bridge network.

Signing up for MongoDB and working from VS Code

We'll be using MongoDB as the database for our microservice application. We could install MongoDB locally as a Docker container like RabbitMQ and Redis. But MongoDB is also available as a (free) cloud database, and we prefer this option because it's cloud-native.

These are the main steps for signing up, which can change over time due to website updates by MongoDB:

1. Navigate to `https://www.mongodb.com/cloud/atlas/register` and sign up.
2. When asked for which subscription you want, choose the free option.
3. Answer the questions about how you'll use MongoDB.
4. When the **Security Quickstart** page opens, create a database user:
    - In the **Username** field, enter `django-microservice`.
    - Leave the password as suggested, but *copy* and *store* the password because we'll need it later when we connect from our microservices to a database.
    - Click **Create User** to make the database user.
5. Select **Cloud Environment** as the place where you would like to connect from.
6. Click **Finish and Close**.

Working with MongoDB from VS Code

You're signed up now to MongoDB and have access through the `django-microservice` account. In *Chapter 4*, *Cloud-Native Data Processing With MongoDB*, we'll further look into MongoDB. For now, we'll end by installing and configuring the VS Code extension for MongoDB since it allows us to manage databases from VS Code.

To set up MongoDB with VS Code, we first need to find out the connection string for our MongoDB environment:

1. Click **Database** from the left menu in MongoDB's web interface.

   * *The* **Database Deployments** *page opens.*

2. Click **Connect**.

   * *The* **Connect to Cluster** *pop-up opens.*

3. Click **MongoDB for VS Code**.
4. Copy the connection string from item 3 and store the string for later usage.
5. Close the pop-up.

Now, move over to or open VS Code and install the MongoDB extension as follows:

1. Search for the **MongoDB for VS Code** extension and install it.

   * *When the installation is ready, you should see a MongoDB item in VS Code's activity bar.*

2. Click the **MongoDB** item from the activity bar.
3. Unfold the **CONNECTIONS** section.
4. Click the + sign beneath **CONNECTIONS** to add a Mongo DB connection.

   * *VS Code opens a MongoDB pane.*

5. Click **Connect** (Connect with `Connection` string) inside the MongoDB pane.

   * *VS Code opens a pop-up where you can enter the connection string.*

6. In the pop-up, enter the connection string you copied earlier and replace the password (`<password>`) segment with the password for the `django-microservice` database user.
7. Type *Enter* to confirm the connection string.

   * *The MongoDB extension shows your database connection underneath the* **CONNECTIONS** *section with the* **connected** *status.*

8. Click the database connection to unfold its content.

   * *The treeview shows the* `admin` *and* `local` *databases, which MongoDB created by default. In Chapter 4, Cloud-Native Data Processing With MongoDB, we'll create our own database.*

Okay, that's it. We have everything in place now. And we're ready to start building and kick off by looking into the requirements for the sample microservices application we're asked to develop.

Analyzing the sample microservices application

Throughout the rest of this book, we'll explore developing Django microservices in detail and put that into practice by creating a sample microservices application.

Regarding that sample application, the *Product Owner* of the *Subscription Management team* of a computer magazine publisher asked us to develop a new web app for registering subscriptions.

The present app registers all new subscriptions without checking if a newly entered address already exists in the address list. This causes the same addresses to appear multiple times in the list with slightly different spelling. For example, the address `East River Street 14` occurs in these variants:

- `East River Street 14`
- `East river street 14`
- `East River str. 14`
- `East river str. 14`
- `Eest River street 14`
- `East Rover street 14`

The first four variants differ in capitalization and abbreviation. The last two variants are due to typos, but the current app accepts them all. As a result, the address list is cluttered and contains thousands of unnecessary addresses, making retrieval slow.

The Product Owner wants the new app to check if a newly entered address matches existing addresses and only add a new address if it doesn't match enough with an existing address. If the new address matches an existing one, the app must assign the best-matching existing address to the subscription.

The Subscription Management team maintains the address list to ensure the correct base addresses.

Matching an address

Let's simulate the matching in the new app to see how it should work. A new subscriber enters the address `Guido van rossem street 101`, where the address list contains the base address `Guido van Rossum Street 101` because that's the correct name spelling of Python's creator.

Now, the new app should try to match the entered address with a base address, and in this case, the base address "Guido van Rossum Street 101" should be registered for the subscription. To do so, we'll use *fuzzy string matching*, a technique of finding strings that partially match a given string.

Fuzzy string matching calculates the differences between strings with the *Levenshtein Distance metric*. The result is a score between 0 and 100, where 0 is no match at all, and 100 is a complete match.

Matching "Guido van rossem street 101" with "Guido van Rossum Street 101" gives a score of 89, which is enough for us to match. So, we don't add a new address and assign the existing base address to the new subscription.

The app's requirements as user stories

The Product Owner wrote out the app's requirements as user stories. This is the story from the new subscriber's perspective:

As a new subscriber, I want to register for a subscription to a computer magazine so that I'll receive the magazine monthly at my specified address.

And this is the user story for address matching from the Subscriber Management team's perspective:

As the Subscription Management team, we want newly entered subscription addresses to be matched with existing base addresses so that we'll register correct and unique addresses only.

And this is the user story to confirm a new subscription from the Subscriber Management team's perspective:

As the Subscription Management team, we want to confirm a new subscription by email so that the subscriber knows the subscription has succeeded and can verify the delivery address.

If we set this out in a diagram, the new subscription flow will look like this:

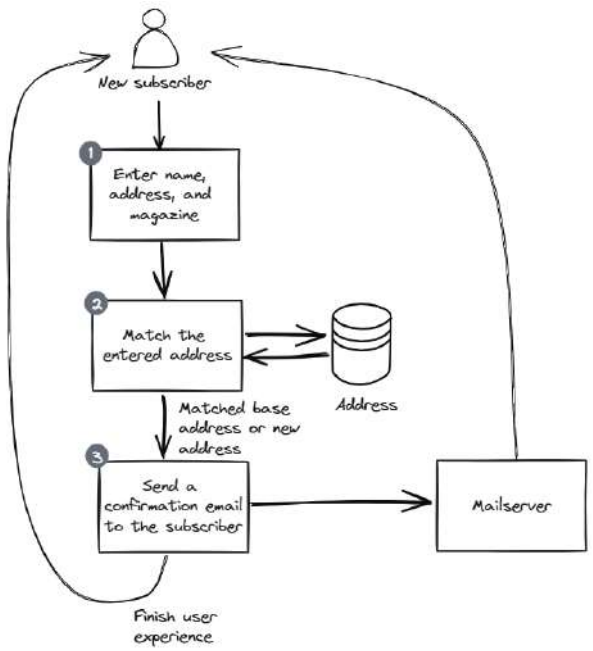

Figure 3.3 – The new subscription flow

When discussing the part for matching addresses, we discover that the present list contains over 90,000 base addresses and that a fuzzy text matching requires approximately 4 seconds to match a new address. The Product Owner indicates that it's unacceptable if the new subscriber has to wait for this matching execution during the user experience.

The same applies to sending the confirmation email, which takes at least 5 seconds to complete. This makes the address matching part and the email sending part perfect candidates for building them as microservices. Because with a microservices setup, the app only has to collect the subscriber's data, offload the tasks for matching and email sending, and finish the user experience almost instantly.

However, there is one caveat in this microservices scenario because the email-sending task depends on the (result of the) address-matching task. So, offloading both tasks by the app will not work because both tasks work independently in parallel, and the email-sending task wouldn't have the matched delivery address.

We solve this by chaining the address-matching and the email-sending task, where the app is the producer that offloads the address matching to a matching worker and where the matching worker, in turn, acts as the producer that offloads the email sending. The microservices-based flow then looks like this:

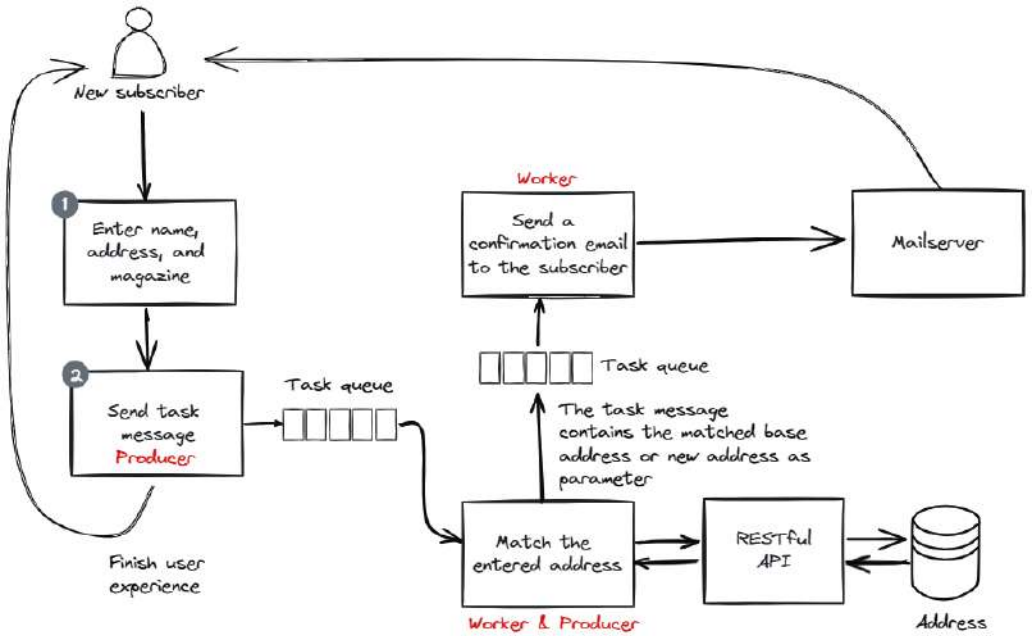

Figure 3.4 – The new subscription microservices architecture

In this setup, we have these microservices:

- The address-matching microservice.
- The RESTful API that delivers and adds addresses.
- The email-sending microservice.

Furthermore, we have this division of producer and worker roles:

- The app acts as the producer for the address-matching worker.
- The address-matching worker calls and interacts with the RESTful API.
- The address-matching worker acts as the producer for the email-sending worker.

This way, the app only needs to offload the matching task and finish the user experience. And because the matching worker offloads the email-sending task after receiving the address from the API, the email worker gets the necessary input. Meanwhile, as the microservices require, the matching and email worker still operate independently. Whereby we fulfill the requirements from the user stories.

Splitting the requirements into use cases

Finally, we derive use cases for the microservices from the requirements, and this is the use case for the address-matching microservice:

Overview	Match a newly entered subscription address with existing addresses.
Actor	A microservice written in Python.
Trigger	A new task arrives in the matching task queue.
Steps	1. Parse the task message. 2. Load the existing base addresses from an API in a data frame. 3. Match the new address with the address data frame and filter the address with the highest score. 4. Is the score of the filtered address higher than the minimum score? I. Yes: Set the filtered address as the delivery address. II. No: Add the originally entered address to the database and set this address as the delivery address. 5. Offload the email-sending task.
Outcome	The delivery address for confirmation.

Table 3.2 – Address matching use case

The use case for the RESTful API looks like this:

Overview	Retrieve registered base addresses and add new addresses.
Actor	A RESTful API with **Django REST Framework** (**DRF**).
Trigger	A request for retrieving addresses or creating a new address.
Steps	1. Parse the request. 2. Query the database. 3. Compose the response. 4. Return the response.
Outcome	A data set with base addresses or a new address added.

Table 3.3 – RESTful API use case

Practically, this means that our API will have an endpoint like this: `https://.../api/v1/addresses/`.

> **Important note**
>
> Our matching microservice and the RESTful API will process the entire address list for each new subscription. There are more efficient approaches than this, and we probably design this differently in our daily developer lives. For instance, we could only load addresses within a specific postal code or area. But here, we leave it as it is because it stresses the time aspect of microservices.

And lastly, this is the use case for the email-sending microservice:

Overview	Send a confirmation email to the subscriber.
Actor	A microservice written in Python.
Trigger	A new task arrives in the task queue.
Steps	1. Parse the task. 2. Prepare the email. 3. Open a connection to the mail server. 4. Send the email request to the mail server.
Outcome	The subscriber receives the email.

Table 3.4 – Email sending use case

This is enough to get us started developing. We'll work out further details as we develop the application.

Phasing the development of the sample application

We break down the development across the microservices architecture's components. Divided into several chapters, we follow this phasing in *Part 2*, *Building the Microservices Foundation*:

1. Creating the data layer in *Chapter 4*, *Cloud-Native Data Processing With MongoDB*.
2. Building the Address RESTful API in *Chapter 5*, *Creating RESTful APIs for Microservices*.
3. Developing the Django app and microservices in *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ*.
4. Testing and deploying our application in *Chapter 7*, *Testing Microservices*, and *Chapter 8*, *Deploying Microservices With Docker*.

This is great: a complete Django microservices application up and running in four steps. Let's quickly summarize this chapter and start our building journey.

Summary

This chapter taught us how to set up our development and runtime environment for building Django microservices. We paid special attention to Windows developers, who need WSL on their workstations to develop with the Ubuntu Linux distribution.

When the Windows developers caught up, we installed the necessary Python packages and additional software like RabbitMQ and Redis for message brokering. Finally, we analyzed the sample microservices application, which we'll develop throughout this book.

This closes *Part 1*, *Introducing Microservices and Getting Started*, and sets us up for the next chapter, where we lay the data processing foundation for Django microservices with a MongoDB cloud-native database.

Part 2: Building the Microservices Foundation

In this part, you'll learn how to develop Django microservices. First, you'll build the data layer with a cloud-native database and a RESTful API to serve the data processing for microservices. Then, you'll learn how to create microservices based on the Celery and RabbitMQ task queue managers. After that, you'll master how to test microservices. Finally, you'll learn how to containerize and deploy microservices with Docker.

This part contains the following chapters:

- *Chapter 4*, *Cloud-Native Data Processing with MongoDB*
- *Chapter 5*, *Creating RESTful APIs for Microservices*
- *Chapter 6*, *Orchestrating Microservices with Celery and RabbitMQ*
- *Chapter 7*, *Testing Microservices*
- *Chapter 8*, *Deploying Microservices with Docker*

4

Cloud-native Data Processing with MongoDB

Data and data processing are at the heart of any application, including microservices applications like the one we'll develop. And in this, databases play a crucial role as they store and process application data. We have a range of database choices, but we'll apply MongoDB because it's cloud-native like our microservices will be.

In this chapter, you'll learn about MongoDB and address questions like how you set up MongoDB and create a database. Furthermore, you'll learn about NoSQL databases like MongoDB and how they differ from relational databases. You'll also take an advance on developing RESTful APIs and learn how the **Create**, **Read**, **Update**, **Delete** (**CRUD**) operations map to HTTP methods like the `GET` method for querying APIs.

By the end of this chapter, you know how to lay the data processing foundation for cloud-native Django microservices with MongoDB. Including applying CRUD operation to maintain your application data.

To accomplish this, this chapter covers the following topics:

- Introducing MongoDB and cloud-native databases
- Setting up MongoDB
- Creating a Database
- Mapping CRUD operations to HTTP methods

This is where it starts to get really interesting for us developers as you're asked to roll up your sleeves and dig in. So get ready and follow along.

Technical requirements

Depending on your local setup, you may need to add your workstation's IP address to your network's whitelist to work with MongoDB from VS Code.

You can find the code samples for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter04`.

Introducing MongoDB and cloud-native databases

We already peeked into the cloud version of MongoDB in the *Chapter 3*, *Setting Up the Development and Runtime Environment*. There is also a community server version, which we can run on-premise (even in a Docker container). Still, we focus on the cloud version because it coheres nicely with our cloud-native microservices.

In the following sub-sections, we'll look at cloud-native and NoSQL databases and why we would use them.

What are cloud-native databases?

Cloud-native databases are just what their naming implies: databases designed to run in cloud environments. So, instead of running a database on a local server, we connect to a remote cloud server through the internet.

Why use a cloud-native database?

This raises the question of why we would use a cloud-native database, and the answer to that lies in these benefits of cloud-native databases:

- **Scalability**: Cloud-native databases can easily scale to handle different workloads.
- **Resilience**: Cloud-native databases are resilient to failures.
- **Flexibility**: Cloud-native databases can handle structured and unstructured data.
- **Containerization**: Cloud-native databases are often deployed within containers.
- **Microservices integration**: Cloud-native databases are well-suited for microservices architectures.

This makes cloud-native databases very suitable as a data platform for microservices, hence our choice for MongoDB.

MongoDB is a NoSQL database

Another interesting aspect of MongoDB is that it's a **NoSQL database**. NoSQL databases store data in free-format **collections** of **documents**, contrary to relational databases, which store data in structured tables with records.

MongoDB vocabulary

Because MongoDB is a NoSQL free-format database, it uses some terms other than the ones you probably know from relational databases. The most common MongoDB terms correspond to their counterparts in relational as follows:

MongoDB	**Relational databases**
database	database
collection	table
document	record
field	field
index	index

Table 4.1 – MongoDB vocabulary and RDBMS counterparts

The main difference between MongoDB and relational databases is in collections and documents, which makes sense because MongoDB is a free-format database, as we'll see in the following subsection.

MongoDB is a free-format database

MongoDB doesn't require a formal table and record structure but allows flexible data storage. For instance, if you store customer data, one customer can have fields that other customers don't like in this example:

Figure 4.1 – Customer documents from the MongoDB sample dataset

The first customer document is fully expanded and contains an `account` array and `tier_and_details` objects, again having a `benefits` array. Such flexible data structures are typical for NoSQL databases. They can significantly improve data processing performance because we don't need to join a sequence of tables as we would in a relational database to get our results. Everything we need is just in one collection.

Furthermore, as we see in the second document, documents can contain different fields. Did you spot the difference? The second document doesn't have an `active` field beneath the `email` field, and that's perfectly fine in a NoSQL database.

Loading the sample data

You can also load the sample data, including the customer document examples, into the MongoDB instance you opened in the previous chapter. Just follow these steps:

1. Navigate in your browser to the MongoDB web interface at `https://cloud.mongodb.com`.
2. Click **Database** from the left menu.
3. Click the **...** button (the button after the **Browse Collection** button) and select **Load Sample Dataset**.

   * *MongoDB adds the sample databases and collections to your cluster and signals after a while that it has finished loading.*

   You can browse the sample data by clicking **Browse Collections**, which lists the added databases. Now, try to find and open the `customers` collection within the `sample_analytics` database. Expand some documents to check their content. Also, browse through the other databases and their collections to get an idea of how to store data in MongoDB.

Alternatively, walk through the sample dataset from VS Code through the MongoDB extension we installed in the previous chapter. The MongoDB extension shows the documents in JSON format.

There's really no SQL in NoSQL databases

Data retrieval in a NoSQL database also differs from relational databases, as NoSQL databases don't work with SQL. Nevertheless, MongoDB collaborates with Django ORM, and in the subsection, *Mapping CRUD operations to HTTP methods*, we'll look at an example of how to query a MongoDB collection from a Python script.

But first, we'll set up MongoDB so we can work with databases and collections.

Setting up MongoDB

At the highest level, MongoDB organizes databases in projects, and within a project, you have one or more clusters with one or more databases. During the signing up for MongoDB in the *Chapter 3*, *Setting Up the Development and Runtime Environment*, we've already created a free cluster for our microservices application.

To set up MongoDB, we'll start with the option to create a paid cluster. In the consecutive sections we'll create a database user and set up MongoDB for Django.

Optional: creating a paid cluster for production databases

A free cluster is fine for a learning project like ours, but you'll need a paid serverless or dedicated cluster for a production database. If you want a production cluster, take these steps to create one:

1. Click **Database** from the left menu.
2. Click **+ Create**.

   * *The* **Create New Cluster** *page opens.*

   From here, you have two choices:

   - **Serverless**: pay per operation with a fixed amount per 1M reads.
   - **Dedicated**: pay as you go, where you pay per hour.

   After making your selection, you can select a cloud provider and region. And you can configure additional settings like the backup you want.
3. Finally, click **Create Instance** to finish the new cluster.

Creating a database user

While signing up for MongoDB, we already created a database user through the *Security Quickstart* step. But we can also add database users later. So, let's create a new user with read-and-write access for all databases within our cluster:

1. Click **Database Access** from the left menu.
2. Click **+ Add New Database User**.

   * *MongoDB opens the* **Add New Database User** *pop-up form.*
3. Leave the *Authentication Method* to **Password**, and in the user name field, enter `django-restful-api`.
4. Click **Autogenerate Secure Password**.
5. Copy and store the password for later reference.
6. Click **Add Built In Role**.

   * *The button makes way for the* **Select Role** *combo box.*
7. In the *Select Role* list, select **Read and write to any database**.
8. Click **Add User**.

MongoDB is now in place and ready to connect to and work with Django.

Setting up our MongoDB cluster for Django

MongoDB also collaborates with Django ORM. For that, we need to adjust the Django settings of our project. To do so, move over to the terminal shell where you created the `django-microservices` directory and follow these steps:

1. Create the Django project for the subscription application we'll develop:

```
$ django-admin startproject subscription_registration
```

2. Enter `code .` to open `django-microservices` directory in VS Code.
3. In VS Code, open the file `settings.py` from the `subscription_registration` subdirectory.
4. Replace the `Database` section in `settings.py` with the following section:

```
DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'Subscription',
        'ENFORCE_SCHEMA': False,
        'CLIENT': {
                'host': '<connection_string>'
        }
    }
}
```

`ENGINE` refers to the `djongo` Python package we installed in the previous chapter. And `NAME` refers to the name of our database, which we'll create in the next subsection, *Creating a Database*.

For now, replace the `<connection_string>` placeholder with the connection string you stored in the *Chapter 3*, *Setting Up the Development and Runtime Environment*, which looks something like this:

```
mongodb+srv://<username>:<password>@cluster0.yjchpyg.mongodb.
net/
```

You can also recall the connection string from the MongoDB web interface by selecting **Database | Connect | MongoDB for VS Code** and copying the connection string.

Within the connection string, replace the `<username>` placeholder with `django-microservice` and `<password>` with the password you stored in the previous chapter.

5. Finally, save `settings.py`.

These steps enable MongoDB for Django ORM, so we can make Django migrations and migrate them to our MongoDB database once we have defined our Django models.

But for that, we need a database that we create next.

Creating a Database

Databases are the heart of any database system, and MongoDB is no exception. MongoDB's setup with projects, clusters, and databases is meant to organize our data layer, and databases are central to that. Creating a database takes the following steps from the MongoDB web interface:

1. Click **Overview** from the left menu.
2. Click **Add Data Options**.

   * *The* **Add Data Options** *page opens.*
3. Click **START** inside the **Create Database on Atlas** card.
4. In the **Database name** field, enter *TemporaryDB*.
5. In the **Collection name** field, enter *Products*. Collections are MongoDB's equal to tables.
6. Click **Create Database**.

   * *MongoDB creates the database and the collection and shows the find/query page for the* `TemporaryDB.Products` *collection.*

You've created the database and can start storing data in collections and documents.

Creating documents inside a collection

Usually, Django applications will add data to MongoDB, but let's add a product from the interface to see how that works:

1. Click **Insert Document**.
2. In the *Insert Document* pop-up, you're completely free to define your document. So in *line 2*, you can enter something like:

   ```
   "product_id": "bz-906-f16"
   ```
3. To add an extra line, hoover your cursor over *line 2* and click **+**, followed by **Add field after**.
4. And enter content like this in *line 3*:

   ```
   "product_name": "Django T-shirt, size L"
   ```

   But you can also make up your own content since a NoSQL database like MongoDB leaves you free.
5. When you're ready, click **Insert**.

   * *MongoDB adds the document, including an auto-generated* `_id` *element.*

Now, add another product in the `Products` collection with a different field layout. Easy and flexible, right?

Updating documents

To change an existing document, take the following steps:

1. Hover over the document you want to change and click the **pencil** symbol.

   * *The document opens in edit mode.*

2. Make the necessary changes.
3. Click **Update** to store the modified document.

Only one data manipulation operation left, deleting, which we'll explore next.

Deleting documents and collections

Removing a document is easy as well from the web interface:

1. Hover over the product document you have inserted and click the *bin* button.
2. Click **Delete**.

And removing a collection requires only a few steps, too:

1. Hover over the **Products** collection underneath the **TemporaryDB** database and click the **recycle bin** button.
2. In the **Enter collection name** field, enter *Products*.
3. Click **Drop**.

   Since `Products` was our only collection, MongoDB also removed the `TemporaryDB` database.

Okay, you now know how to create and maintain a MongoDB database. Let's finish this section by creating the database for our microservices application. Make a database named `Subscription` with a collection named `Dummy` inside your cluster through the web interface. We only need the `Dummy` collection to create the database, as we'll create the necessary collection from Django in the next subsection.

Mapping CRUD operations to HTTP methods

Basically, data processing is about creating, reading, updating, and deleting data as it converges in the *CRUD* acronym. The address-matching microservice from our sample microservices application will add addresses to the `Subscription` database. We have two options to set up the microservice for this:

1. Let the microservice update the database directly through the `pymongo` Python package or Django ORM.
2. Engage a Web API, which updates the database, and let the microservice call that Web API.

Since Django ORM is easier to apply than a *pymongo – MongoDB* connection, we prefer Django ORM within option 1. We still choose option 2, the Web API, because a Web API better aligns with the microservices architecture where the components should be self-contained, single-tasked, and independent.

Choosing the Web API option means we'll build a RESTful API with DRF to handle the data processing. And this requires us to map CRUD operation to HTTP methods because HTTP methods are the access roads to RESTful APIs. The following table shows an overview of CRUD operations and the corresponding HTTP method(s):

CRUD operation	**HTTP method**
Create	`POST`
Read	`GET`
Update	`PATCH`: update individual fields of an existing record `PUT`: overwrite an entire existing record
Delete	`DELETE`

Table 4.2 – Mapping CRUD to HTTP methods

For example, to create a product through a fictional Web API, we could have an HTTP `POST` request in `curl` like this:

```
$ curl -d '{"product_id":"HJ-10W-H96",  "product_name":"Django Cap"}'
-H "Content-Type: application/json" -X POST http://api.django-merch.
com/v1/products/
```

You probably know **curl**, but it stands for **client URL** and is a command-line tool to exchange data with a Web API.

To read all products, we cloud issue a curl command for an HTTP `GET` request like this:

```
$ curl -X GET http://api.django-merch.com/v1/products/
```

Despite our choice to build a Django RESTful API with Django ORM for our application, we'll also explore a Python script based on a pymongo – MongoDB connection. Because this gives us insight into how CRUD operations work from Python to MongoDB at the lowest level.

CRUD operations on MongoDB with Django ORM

In *Chapter 5*, *Creating RESTful APIs for Microservices*, we'll develop the complete RESTful API to serve our microservices application. Here, we limit ourselves to a simple Django app, which performs the *Create* operation from CRUD for addresses through Django ORM.

The app will have a form to enter an address and has this directory structure:

```
subscription_registration/
|
├── subscription/
|   ├── templates/subscription
|   |   ├── base.html
|   |   ├── subscription.html
|   |   └── success.html
|   |
|   ├── forms.py
|   ├── models.py
|   ├── urls.py
|   └── views.py
|
├── subscription_registration/
|   ├── urls.py
|   └── settings.py
|
└── manage.py
```

In the following subsections, we first create the app and the model, and then we'll make the form and related parts like templates and the view. And, very importantly, we'll test the app.

The app and the model

We'll begin by adding a `subscription` app to our Django project. Be sure you're in the upper `subscription_registration` directory when adding the app:

```
$ python3 manage.py startapp subscription
```

Next, add the app to `settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'subscription',
]
```

And specify the `Address` collection through a `models.Model` based `class` in `subscription/models.py`:

```
from django.db import models

class Address(models.Model):
    name = models.CharField(max_length=120)
    address = models.CharField(max_length=120)
    postalcode = models.CharField(max_length=20)
    city = models.CharField(max_length=120)
    country = models.CharField(max_length=80)
    email = models.EmailField()
```

Next, prepare the migration:

```
$ python3 manage.py makemigrations subscription
```

And migrate to MongoDB:

```
$ python3 manage.py migrate
```

Now, check the web interface to see if Django created the `subscription_address` collection in the `Subscription` database. If you still have the MongoDB session open, you might need to refresh the **Collections** page to see the new collection. Besides the `subscription_address` collection, Django also added its standard data sources like `auth_user` and `django_migrations`.

The form, templates, view, and URLs

Okay, the data connection between Django ORM and MongoDB works. This enables us to complete the `subscription` app. We start by specifying the subscription form and create a `forms.py` file inside the `subscription` directory with this code:

```
from django import forms

class SubscriptionForm(forms.Form):
    name = forms.CharField(label="Your name")
    address = forms.CharField(label="Address")
    postalcode = forms.CharField(label="Postal code")
    city = forms.CharField(label="City")
    country = forms.CharField(label="Country")
    email = forms.EmailField(label="Email")
```

This is all we need for the subscription web page we'll create next. To do so, create a `templates/subscription` directory inside the `subscription` directory. And inside the `templates/subscription` directory, create this `base.html` file:

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
     initial-scale=1.0">
    <link rel="stylesheet"
          href="https://unpkg.com/bamboo.css">
    <title>Subscription</title>
</head>

<body>
    {% block content %}
    {% endblock content %}
</body>
</html>
```

This straightforward HTML base file styles our pages with *bamboo CSS*.

Next, we'll create the `subscription.html` file, which holds the form from `forms.py` for entering subscriptions:

```
{% extends 'subscription/base.html' %}

{% block content %}

<h2>Enter the address where we can deliver your magazine</h2>

<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Subscribe" />
</form>

{% endblock content %}
```

Again, no surprises here, as we invoke the form in a Django standard way. Finally, we want to show a success page when a user correctly enters a subscription. So, create a `success.html` file with this code:

```
{% extends 'subscription/base.html' %}

{% block content %}

<h1>Thanks!</h1>

<p>You'll receive the latest edition of our magazine within three
days.</p>

{% endblock content %}
```

Excellent, all we need now is a `views.py` file to spin up the form and two `urls.py` files to expose the form. Add this code to a `views.py` file:

```
 1 from django.views.generic.base import TemplateView
 2 from django.views.generic.edit import FormView
 3 from subscription.forms import SubscriptionForm
 4 from subscription.models import Address
 5
 6 class SubscriptionFormView(FormView):
 7    template_name = "subscription/subscription.html"
 8    form_class = SubscriptionForm
 9    success_url = "/success/"
10
```

```
11    def form_valid(self, form):
12      address = Address(name=form.cleaned_data["name"],
                          address=form.cleaned_data["address"],
                 postalcode=form.cleaned_data["postalcode"],
                             city=form.cleaned_data["city"],
                          country=form.cleaned_data["country"],
                              email=form.cleaned_data["email"])
13      address.save()
14
15      return super().form_valid(form)
16
17 class SuccessView(TemplateView):
18    template_name = "subscription/success.html"
```

Lines 7 and 8 specify the subscription form and template.

Line 9 and *lines 17-18* set the success page, which displays when the form is valid.

Speaking of the valid form status, *line 11* defines the `form_valid` method within the `SubscriptionFormView`.

Within the `form_valid` method, *line 12* creates a new `address` object for a correctly entered subscription address. Finally, *line 13* saves the new address to our MongoDB address collection.

Then the last part, the URL files. First, make a `urls.py` file inside the `subscription` directory with the following code:

```
1 from django.urls import path
2 from subscription.views import SubscriptionFormView,
                                  SuccessView
3
4 app_name = "subscription"
5
6 urlpatterns = [
7   path("subscription/", SubscriptionFormView.as_view(),
          name="subscription"),
8   path("success/", SuccessView.as_view(),
          name="success"),
9 ]
```

Line 2 imports our views, and *lines 7-8* expose them as URL paths. For example, the subscription address page will be accessible through this URL if we run django locally: `https://127.0.0.1/subscription`.

At least if we include the subscription URL in the Django project URLs file. To do so, open the `urls.py` file from the `subscription_registration` directory and set its code to:

```
1 from django.contrib import admin
2 from django.urls import include, path
3
4 urlpatterns = [
5    path("admin/", admin.site.urls),
6    path("", include("subscription.urls")),
7 ]
```

Line 6 now includes the URLs from `subscription/urls.py`.

Phew, we did a lot of work. But the app and the address form are all set now and waiting to be tested in the next subsection.

Testing the application

It's time to check if everything works. Run the Django project and open the URL `https://127.0.0.1/subscription` in your browser. If you see this page, you're halfway:

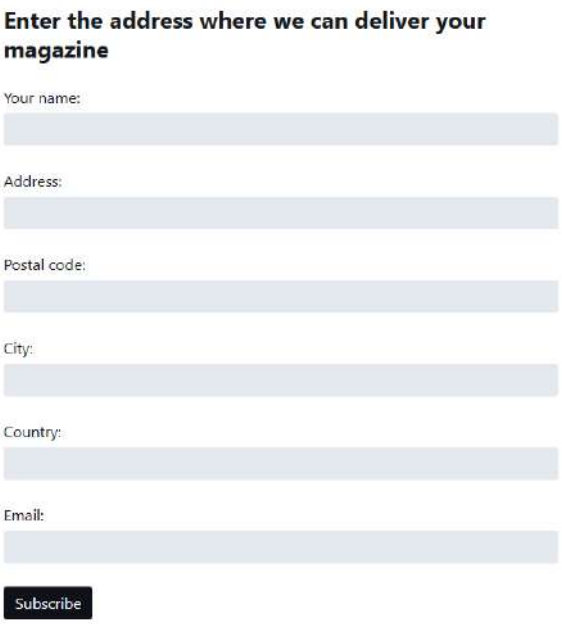

Figure 4.2 – The subscription address page

Now, enter some data and click **Subscribe**. You should see the success page almost instantly, thanking you for subscribing. Everything looks fine, but we need to verify our Django app has added the address to MongoDb. Move to the MongoDB Web interface, refresh the **Collections** page, and inspect the document with your entered address.

Superb, with this, we set the data layer that we need for the RESTful API we'll build in *Chapter 5*, *Creating RESTful APIs for Microservices*.

Practically, this is what we need to know about Django – MongoDB data processing for building Django microservices applications. But as developers, we like to understand concepts at the lowest level, and if you do, follow along with the next subsections, where we'll look at the various MongoDB CRUD operations with the `pymongo` Python package.

CRUD operations on MongoDB with pymongo

Okay, let's dig into how Python and `pymongo` collaborate with MongoDB. This collaboration is a four-stage process in Python:

1. Set up a MongoDB client that communicates with our MongoDB cluster.
2. Connect to the MongoDB database we need through the MongoDB client.
3. Link to the necessary collection or collections.
4. Perform data operations on the collections.

To bring these stages together, we step outside Django for a moment and create four Python scripts covering the CRUD operations.

Start with creating a `pymongo-mongodb` directory inside our `django-microservices` directory. And ensure that we have activated the virtual environment.

Creating

To create an address document, make a file called `create_address.py` inside the `pymongo-mongodb` directory and set its content to:

```
1 from pymongo.mongo_client import MongoClient
2 from pymongo.server_api import ServerApi
3
4 con = "mongodb+srv://django-microservice:
        <password>@<cluster>/?retryWrites=true&w=majority"
5 client = MongoClient(con, server_api=ServerApi('1'))
6 db = client["Subscription"]
7 col = db["subscription_address"]
8
9 address = {"name": "Monthy Python",
```

```
                "address": "Hardwood Lane",
                "postalcode": "MP1"
                "city": "London",
                "country": "England"
                "email": "monthy@python.com"
               }
10 col.insert_one(address)
```

Line 4 specifies the connection string for our MongoDB cluster. replace `<password>` and `<cluster>` with your values.

Line 5 uses the connection string to set up the client.

Line 6 connects to the `Subscription` database, and *line 7* links to `subscription_address` collection. With this, we have the connection and can start working on the data and add a new address.

Line 9 sets the content of the new address as a Python dictionary, and *line 10* inserts it into the `subscription_address` collection.

Now, run this script and check MongoDB to inspect the new address.

Reading

To read all addresses, create a `read_addresses.py` file and add this code to it:

```
 1 from pymongo.mongo_client import MongoClient
 2 from pymongo.server_api import ServerApi
 3
 4 con = "mongodb+srv://django-microservice:
        <password>@<cluster>/?retryWrites=true&w=majority"
 5 client = MongoClient(con, server_api=ServerApi('1'))
 6 db = client["Subscription"]
 7 col = db["subscription_address"]
 8
 9 for address in col.find():
10     print(address["name"])
```

Lines 1-7 are the same as for the creating script. Again, replace `<password>` and `<cluster>` with your values.

Line 9 loops over all documents in the `subscription_address` collection, and *line 10* prints out the address names.

Execute the script to see the names output.

Updating

To update a document, we'll need to select the document to be modified and specify our field updates. Make an `update_address.py` file with this code:

```
1 from pymongo.mongo_client import MongoClient
2 from pymongo.server_api import ServerApi
3
4 con = "mongodb+srv://django-microservice:
        <password>@<cluster>/?retryWrites=true&w=majority"
5 client = MongoClient(con, server_api=ServerApi('1'))
6 db = client["Subscription"]
7 col = db["subscription_address"]
8
9 col.update_one(
     {"name": "Monthy Python"},
     {"$set": {"address": "Liverpool Street"}}
  )
```

Again, *lines 1-7* are the same as in the previous scripts. If we took these CRUD scripts to production, making the connection would be an excellent candidate to include in a Python module. But since this is about explaining CRUD with `pymongo` we'll leave it as it is.

Line 9 does the work here through the `update_one` method, which has two parameters. The first parameter is a query object, which selects the document to update. In our case, we look for the document where the `name` field contains `"Monthy Python"`. The second parameter is an object with the values to update, preceded by the `$set` key. In our case, we update the content of the `address` field with another street.

Run the script and check MongoDB for the updated address.

Deleting

By now, you most likely know how `pymongo` works, and therefore, you're challenged to create a script that deletes the `Monthy Python` address yourself. You can find information about deleting documents in the MongoDB documentation at `https://www.mongodb.com/docs/manual/tutorial/remove-documents/#delete-only-one-document-that-matches-a-condition`.

For your reference or to compare, you can find a sample of the delete script on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter04`.

Cleaning up

We developers like to keep things tidy, so we delete the `Dummy` collection from our `Subscriptions` database in MongoDB.

If you also want to keep your `django-microservices` directory tidy, remove the `pymongo-mongodb` with the CRUD scripts we created. We won't need them anymore, but you can keep them for later reference—your pick. And you can always check the code on GitHub, of course.

Summary

This chapter started by explaining what cloud-native databases are and that we selected MongoDB as the database for our application because the cloud version of MongoDB coheres well with the principles of the microservices architecture.

Next, we learned about NoSQL databases and how they differ from relational databases. This also led us to the vocabulary for MongoDB, where collections are the counterpart of tables.

Then, we prepared MongoDB for our application and created our first database and collection. Finally, we mapped the CRUD operations and their counterpart HTTP methods in anticipation of the RESTful API we'll build for processing the data in our application.

With this, we have completed the data layer of our application, and we're set for the next chapter, where we'll learn about developing RESTful APIs with DRF.

5
Creating RESTful APIs for Microservices

Application programming interfaces (**APIs**) play an essential role in software development because they support loose coupling of software components, improving factors such as the maintainability and robustness of applications. This also applies to microservices applications where APIs are key components to meet microservices principles such as being self-contained and resilient.

APIs come in different forms, but we'll focus on **REpresentational State Transfer** (**RESTful**) APIs because they are microservices in themselves and fit well in the microservices architecture.

In this chapter, you'll learn what RESTful APIs are and how you can build them with Django and the **Django REST Framework** (**DRF**). You'll also learn how to implement **Create**, **Read**, **Update**, **Delete** (**CRUD**) operations and corresponding HTTP methods with DRF.

By the end of this chapter, you'll know what RESTful APIs are and how you consume such APIs with HTTP methods such as `GET`, `POST`, and `DELETE`. Furthermore, you'll be able to build a RESTful API with DRF, including handling errors.

Therefore, this chapter addresses the following topics:

- Introducing RESTful APIs
- Building RESTful APIs with DRF
- Error handling

This is where you start programming and will develop the first microservice for the sample application. Follow along to take your next steps in building state-of-the-art Django microservices applications.

Technical requirements

You can find the code samples for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter05`.

Introducing RESTful APIs

The RESTful API architecture decouples software into components that communicate programmatically in a *client-server* model with a *request-response* protocol, like in this example:

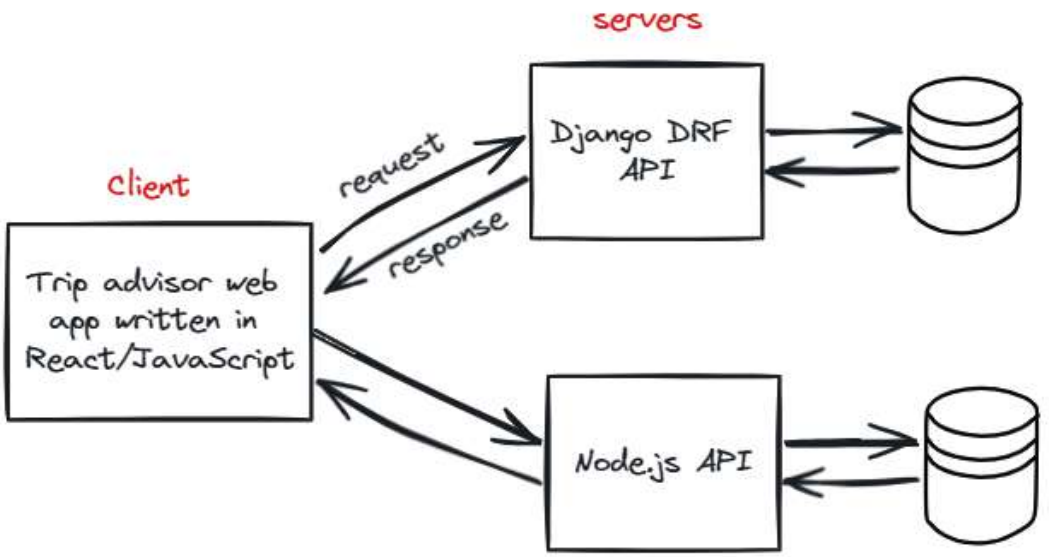

Figure 5.1 – RESTful API architecture for a web application

This example has three software components:

1. A React/JavaScript-based web client, which acts as the client
2. A DRF API, which acts as a server
3. A Node.js API, which acts as a server

The web client and the API servers communicate via the HTTP protocol by sending *requests* and *responses*. A request contains at least an HTTP method such as `GET` and the RESTful API's URL or endpoint, like this example:

```
$ curl -X GET https://api.xyz.com/v1/products/
```

Furthermore, requests can also contain *parameters* to select specific products or a request *body* with data for adding a product.

A response contains at least a *status code* indicating whether the RESTful API processed the request successfully. Furthermore, requests can hold the following:

- The requested data
- An error message

The most common status codes you'll encounter are the following:

- *200 OK*: Successful request
- *201 Created*: Created a new resource
- *204 No Content*: Successful request without returning data
- *400 Bad Request*: The RESTful API can't parse or process the request
- *401 Unauthorized*: The RESTful API requires authentication, but the client isn't providing the (correct) credentials
- *404 Not Found*: The RESTful API can't find the requested resource
- *500 Internal Server Error*: Something went wrong with the RESTful API

You can check `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes` for a complete list of status codes and their meaning.

An example of the data segment of a response could look like this:

```
{"prod_id": "JKL-103", "name": "Django pen", "price": 0}
```

RESTful APIs serialize their response into JSON or XML format in scenarios such as these:

1. The web app sends an HTTP request for trip data to the Django API.
2. The Django API parses the request and executes the corresponding processing to fetch trip data from its data sources.
3. The Django API serializes the fetched data into JSON format and sends the data as a response to the client.
4. The client parses the response and shows the trip data on a web page.

This way, all the technical details of retrieving and serializing data are hidden from the client and handled by self-contained and task-specific API servers that act as gateways. In the same way, the client focuses on the presentation task. All are working independently. If an API doesn't respond, the client can't show trip data, but it won't crash and will still function for the user.

This is a significant advantage of working with RESTful APIs, but they have more benefits, as we'll see in the following subsection.

Benefits of RESTful APIs

Besides being self-contained, RESTful APIs are also the following:

- Scalable because they can run on multiple servers
- Simple because they use the HTTP protocol
- Flexible because they support common data types such as JSON and XML
- Widely adopted
- Platform independent because they can be built in different programming languages and run on different platforms

On the other hand, RESTful APIs also have some challenges, such as version control. But we can overcome those just fine with proper procedures and standard practice. To get everything out of RESTful APIs and to deal with the challenges, it's essential to fully understand what they consist of and how they operate, as we'll explore next.

The RESTful API architecture

RESTful APIs cover the following basic parts:

- **Resources**: The information a client can request where the API gathers the information per entity as a resource such as a customer or product. In the example of *Figure 5.1*, the *trip* is a resource. And for our sample microservices application, the *address* is a resource.
- **Endpoints**: The URLs where the API exposes itself and clients send requests.

And RESTful APIs process these parts to handle requests:

- **HTTP methods**: The methods a client uses to send a request to an API:
  - **GET request**: Retrieve data about a resource
  - **POST request**: Create a resource
  - **PATCH request**: Update a resource partially
  - **PUT request**: Overwrite a resource entirely
  - **DELETE request**: Remove a resource

- **Parameters**: The options clients send to an endpoint to retrieve specific resource information or to authenticate:
  - **Header**: Such as an API security key to authorize a request
  - **Path**: As part of the called URL to address a specific resource
  - **Query string**: As part of the called URL to select a subset of resources
- **Request body**: The JSON or XML objects a client sends to an API as resource data

Let's bring these parts together in some `curl` examples. The first example requests a list of all hostels in the *Tripadvisor* system:

```
$ curl -X GET https://api.trip.com/v1/hostels/
```

This request applies a `GET` request to access the `api.trip.com` endpoint and addresses the `hostels` resource.

The second example requests the details of a hostel with ID number 24:

```
$ curl -X GET https://api.trip.com/v1/hostels/24/
```

This request adds `24/` as a path parameter to select a specific hostel.

The next example selects hostels that allow dogs:

```
$ curl -X GET https://api.trip.com/v1/hostels/?dogs=yes
```

This request adds `?dogs=yes` as a query parameter.

The last example makes a reservation:

```
$ curl -d '{"hostel_id":24, "start":"2024/03/01", "end":"2024/03/06"}'
-H "Content-Type: application/json" -X POST http://api.trip/v1/
hostels/
```

This request applies a `POST` request to send the `{"hostel_id":24, "start":"2024/03/01", "end":"2024/03/06"}` JSON object as the request body with reservation details. Furthermore, the request sends a header parameter to set the content type to `application/json`.

Okay – we now know that RESTful APIs send requests and responses to endpoints through HTTP methods in a client-server model. This should be enough to get us started with building a RESTful API with Django in the next section.

Building RESTful APIs with DRF

DRF is an extension of Django for developing and deploying RESTful APIs. In *Chapter 3*, *Setting Up the Development and Runtime Environment*, we installed DRF through `pip` as a Python package, and in this section, we'll apply DRF to develop a RESTful API for our sample application. DRF provides us with these functionalities:

- The actual API that clients can access through one or more endpoints
- Serializers that collaborate with Django ORM to transform data from data source to JSON and the other way around
- A set of pre-defined views to quickly and easily create API endpoints for CRUD operations

If we transpose this to Django and zoom in on the Django file structure, we'll see that DRF is primarily about these standard Django files:

- `models.py`: Containing the data model for our API
- `views.py`: Containing the API viewset and specifications for CRUD operations
- `urls.py`: Containing the endpoints for our API views

There is also this specific DRF file:

- `serializers.py`: Containing the serialization specification for the model

These files work closely together as the serializer; the view uses the model (the serializer), and the URL file uses the view.

With this in mind, let's start building a RESTful API for our subscription application. For that, we look again at the overview of our sample application to see where the RESTful API sits:

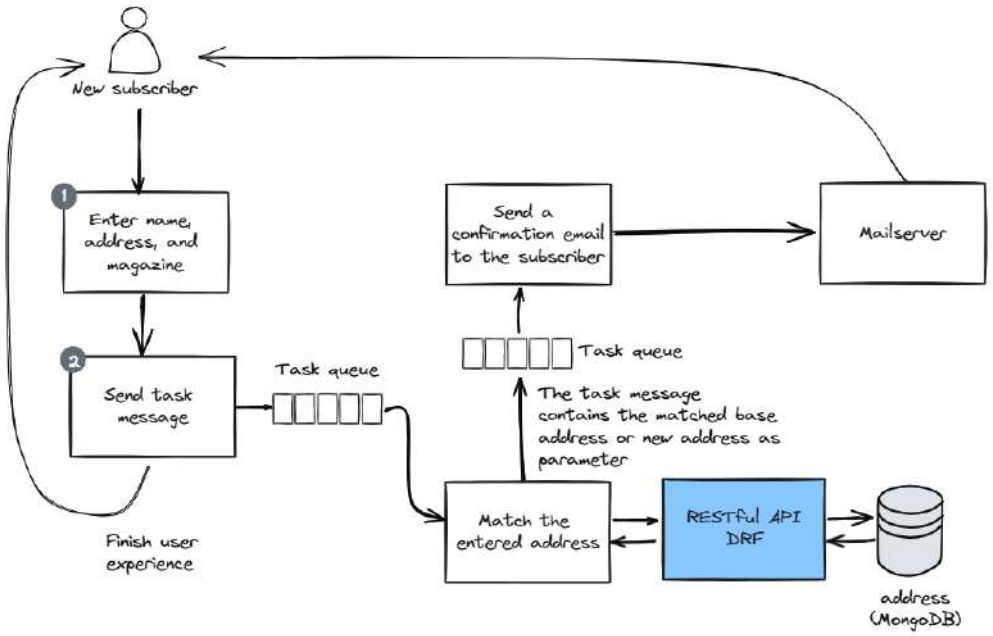

Figure 5.2 – The RESTful API's position in the sample application

The RESTful API will have this directory structure:

```
subscription_apis/
|
├── address_api/
|   ├── models.py
|   ├── serializers.py
|   ├── urls.py
|   └── views.py
|
├── subscription_apis/
|   ├── urls.py
|   └── settings.py
|
└── manage.py
```

And to build the RESTful API, we'll take this route:

1. Setting up DRF
2. Creating a model and a serializer
3. Creating a view and a URL file for the endpoints

Along the way, we'll run, test, and browse our RESTful API.

Setting up DRF

We installed the `djangorestframework` package in *Chapter 3*, *Setting Up the Development and Runtime Environment*, and because of that, we now only need to create a Django project for our RESTful API and set it up for DRF:

1. Inside the `django-microservices` directory, create a Django project for the subscription APIs:

```
$ django-admin startproject subscription_apis
```

2. Create a Django app for the address API:

```
$ python3 -m startapp address_api
```

3. Add the app and DRF to the `settings.py` file:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'address_api',
]
```

   Both the API project and the address API app are ready now. But DRF has a nice feature that lets us view a RESTful API through a browser. This allows us to check if the RESTful API works as intended quickly. To use this feature, we need to add a `path` parameter to the project's `urls.py` file:

```
urlpatterns = [
    ...
    path('api-auth/', include('rest_framework.urls'))
]
```

Our RESTful API will be browsable now, and we continue with the model and serializer.

Creating a model and a serializer

Now, set up this `models.py` file for the address API:

```
from django.db import models

class Address(models.Model):
    name = models.CharField(max_length=120, blank=True)
    address = models.CharField(max_length=120, blank=True)
    postalcode = models.CharField(max_length=20,
                                  blank=True)
    city = models.CharField(max_length=120, blank=True)
    country = models.CharField(max_length=80, blank=True)
    email = models.EmailField(blank=True)
```

This model is very straightforward and is all that DRF needs to serialize address data from MongoDB to JSON and vice versa. To implement the serialization, create a `serializers.py` file in the `address_api` directory with the following code:

```
1 from rest_framework import serializers
2 from .models import Address
3
4 class AddressSerializer(serializers.ModelSerializer):
5    class Meta:
6        model = Address
7        fields = ("id", "name", "address", "postalcode",
                   "city", "country", "email")
```

Line 1 imports the `serializers` functionality from DRF.

Line 2 imports the `Address` class from our models, which must be serialized.

Line 4 declares the `AddressSerializer` class, which is based on `serializers.ModelSerializer`.

Line 6 sets the `model` for the serializer to `Address`.

Line 7 specifies the fields to serialize. In this case, we serialize all fields, but we could specify a subset if we want to.

This is all we need to do to get our data serialized, and from here on, DRF does the actual work. The next step for us is creating a view for our RESTful API.

Creating a view and the URL endpoints

DRF was first released in 2011, and over time, the view mechanism evolved from **function-based** to **class-based views**.

Function-based views are, as their name implies, based on a Python function, which defines the view's behavior. A function becomes a function-based view by decorating it like this:

```
@api_view(['GET', 'POST'])
def address_list(request):
    if request.method == 'GET':
       ...
    elif request.method == 'POST':
       ...
```

The `@api-view` decorator promotes the `address_list` function to a function-based view, which clients can access through the `GET` and `POST` methods. Consequently, the `if ... elif` section handles `GET` and `POST` requests.

Likewise, class-based views are based on DRF-specific Python classes:

```
class AddressViewSet(viewsets.ModelViewSet):
    queryset = Address.objects.all()
    serializer_class = AddressSerializer
```

The generic `ModelViewSet` class integrates all HTTP methods, including `PUT` and `DELETE`, and covers most RESTful API requirements. All we have to do is specify a `queryset` object and a `serializer_class` object for the viewset, and DRF takes care of the rest.

Despite the evolution from function-based to class-based views, both view types still exist today, and their characteristics and differences come down to this:

View type	Benefits	Drawbacks
Function-based	• Simple to develop • Easy to understand • Full control of functionality	• Requires quite some coding • Difficult to extend and reuse • Conditional branching is needed for HTTP methods
Class-based	• Provide built-in processing of CRUD operations • Easy to extend • Simple to reuse	• More difficult to understand because of built-in and implicit logic

Table 5.1 – DRF view-type characteristics

Because of the built-in CRUD operations, which simplify and speed up development, we'll apply class-based views in our sample application. But we'll also look at an example of a function-based view to see how these work.

> **Important note**
> In general, apply class-based views if you can and function-based views if you must because of some specific functionalities offered only by function-based views.

Developing a class-based view

Class-based views have their own evolution, which resulted in the following view types:

- Views based on the **APIview class** for simple operations such as retrieving information
- Views based on the **GenericAPIview class** and **mixins** for CRUD operations
- Views based on **generic class-based views** with the mixins functionality already built in
- Views based on the **ModelViewSet class** with all CRUD operations already on board

Let's create examples of each view type for our sample to see how they work and what they offer. We'll start with a simple `APIview`-based view and end with a sophisticated `ModelViewSet`-based view.

Creating an APIview-based view

`APIView`-based views are the most basic and suitable for simple cases such as returning all addresses from the address collection. We create such a simple `views.py` file this way:

```
 1 from rest_framework.views import APIView
 2 from rest_framework.response import Response
 3
 4 from .models import Address
 5 from .serializers import AddressSerializer
 6
 7 class AddressList(APIView):
 8     def get(self, request, format=None):
 9         addresses = Address.objects.all()
10         serializer = AddressSerializer(addresses,
                                          many=True)
11         return Response(serializer.data)
```

Line 1 imports the `APIView` class.

Line 2 imports the `Response` class that handles the response to the client.

Lines 3-4 import our model and serializer.

Line 7 declares the `AddressList` class that is based on the `APIView` class.

Line 8 defines a `get` function that handles `GET` requests from clients.

Line 9 sets the information to return and *line 10* serializes this information from MongoBD to JSON format.

Finally, *line 11* returns all serialized addresses through a `Response` object to the requesting client.

All we need now is to create a `urls.py` file in the `address_api` directory with this content:

```
1 from django.urls import path
2
3 from .views import AddressList
4
5 ver = 'v1'
6
7 urlpatterns = [
8    path(f'api/{ver}/addresses/', AddressList.as_view()),
9 ]
```

In *line 5*, we initiate a simple version mechanism for our RESTful API. And in *line 7*, we connect the `api/v1/addresses/` endpoint to the class-based `AddressList` view.

To make this endpoint accessible to clients, we need to refer from the project's `urls.py` file to the app's `urls.py` file:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls',
          namespace='rest_framework')),
    path('', include('address_api.urls')),
]
```

Excellent; with this, we've created the first implementation of our RESTful API. Now, we need to test it. Start with running the Django project. Next, create a file called `test_api.py` in the `django-microservices` directory with this test code:

```
1 import requests
2
3 response =
4   requests.get('http://127.0.0.1:8000/api/v1/addresses/')
5 print(response.text)
```

We use the `requests` package to send the request to our RESTful API in *line 4* like a web application or other client would. And *line 5* prints the response text, which should output all the addresses in your `address` collection.

Now, run the test script. Do you see the addresses in JSON format? Great – the RESTful API works. Now, let's extend the view to create new addresses.

With `APIView`-based views, we must define the functionality for CRUD operations in class methods such as `get`. Likewise, we can add a class method that handles `POST` requests to create a new address. To do so, change the `views.py` file:

```
 1 from rest_framework.views import APIView
 2 from rest_framework.response import Response
 3 from rest_framework import status
 4 from .models import Address
 5 from .serializers import AddressSerializer
 6
 7 class AddressList(APIView):
 8     def get(self, request, format=None):
 9         addresses = Address.objects.all()
10         serializer = AddressSerializer(addresses,
                                          many=True)
11         return Response(serializer.data)
12
13     def post(self, request, format=None):
14         serializer =
                  AddressSerializer(data=request.data)
15         if serializer.is_valid():
16             serializer.save()
17             return Response(serializer.data,
                       status=status.HTTP_201_CREATED)
18         return Response(serializer.errors,
                       status=status.HTTP_400_BAD_REQUEST)
```

Adding an address takes some more coding. *Line 3* imports the `status` class because we want to return to the client whether an address was successfully created.

Line 13 declares the `post` method that handles `POST` requests.

Line 15 checks if the address data the client includes in the request is valid.

If the address data is *valid*, *line 16* serializes it and saves it to the `address` collection in MongoDB. *Line 17* returns the created address and status code `201`, which indicates our RESTful API successfully created the address.

If the address data is *invalid*, *line 18* returns a serializer error and status code `400`, which indicates our RESTful API didn't create the address.

To test the added method, change the `test_api.py` file to the following content:

```
 1 import requests
 2
 3 address = {"name": "Neo",
 4            "address": "The Matrix",
 5            "postalcode": "MX702",
 6            "city": "Zion",
 7            "country": "Everland",
 8            "email": "neo@hackers.com"
 9           }
10
11 # CREATE
12 response = requests.post('http://127.0.0.1:8000/
                            api/v1/addresses/',
                            data=address)
13 print(response.status_code)
14 print(response.text)
15
16 # READ
17 response = requests.get('http://127.0.0.1:8000/
                           api/v1/addresses/')
18 print(response.text)
```

Lines 3-9 create a dictionary for the new `address`, which *line 12* sends as the `data` parameter value in the `POST` request to create a new address.

Line 13 prints the status code returned by the RESTful API, which should be `201`. Run the script to see if you get the expected status code. Furthermore, *lines 17-18* print all the addresses, including the newly added one.

Okay, so far, for `APIView`-based views. Now, let's explore how views based on the `GenericAPIview` class and mixins work.

Creating a generic API view and mixins-based view

The next evolution in class-based views is mixins. Mixins classes elaborate on the `GenericAPIView` class and have CRUD operations functionality built in. We still need to define class methods such as `GET` and `POST`, but their implementation is simpler than with the `APIView` class. Change your `views.py` file to the following code to implement mixins for handling all CRUD operations in our RESTful API:

```
from rest_framework import mixins
from rest_framework import generics
from .models import Address
from .serializers import AddressSerializer

class AddressList(mixins.ListModelMixin,
                  mixins.CreateModelMixin,
                  generics.GenericAPIView):
    queryset = Address.objects.all()
    serializer_class = AddressSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

class AddressDetail(mixins.RetrieveModelMixin,
                    mixins.UpdateModelMixin,
                    mixins.DestroyModelMixin,
                    generics.GenericAPIView):
    queryset = Address.objects.all()
    serializer_class = AddressSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def patch(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

To cover all CRUD operations, we'll need two classes. First, `AddressList` covers reading all addresses and creating a new address through the `ListModelMixin` and `CreateModelMixin` mixins in *lines 6-7* with the corresponding `GET` method in *line 12* and the `POST` method in *line 15*.

Second, `AddressDetail` covers reading, updating, and deleting a specific address through the `RetrieveModelMixin`, `UpdateModelMixin`, and `DestroyModelMixin` mixins, with the corresponding `GET` method in *line 25*, the `PUT` method in *line 28*, the `PATCH` method in *line 31*, and the `DELETE` method in *line 34*.

For these methods, we only need to define the response by a `return` statement because the mixins classes already have the functionality to fulfill the corresponding CRUD operation. For example, the `mixins.DestroyModelMixin` class has all the functionality to delete an address.

Because we've added the class-based `AddressDetails` view, we need to add an endpoint for it in the app's `urls.py` file:

```
urlpatterns = [
    path(f'api/{ver}/addresses/', AddressList.as_view()),
    path(f'api/{ver}/addresses/<int:pk>/',
         AddressDetail.as_view()),
]
```

We've added `pk` as a path parameter to select a specific address. `pk` is short for primary key and refers to the address ID, which Django ORM automatically adds when we create a new address.

Okay – it's time to test our RESTful API again. Let's start with overwriting Neo's address with an updated version from the `test_api.py` file:

```
1 import requests
2
3 address = {"name": "Neo",
4            "address": "The Other Side",
5            "postalcode": "1TO",
6            "city": "Zion",
7            "country": "The Matrix",
8            "email": "neo@hackers.com"
9           }
10
11 # UPDATE
12 response = requests.put('http://127.0.0.1:8000/
                            api/v1/addresses/<ID>/',
                            data=address)
13 print(response.text)
```

Line 12 sends a `PUT` request to our RESTful API to overwrite an address entirely. Replace `<ID>` in *line 12* with the ID from the output when you created the address. You can also look up the ID in the MongoDB web interface or list all addresses via a `GET` request.

Next, run the test script and inspect its output, which should show the updated address.

Excellent – now, let's just update a specific field in Neo's address. Comment out *lines 3-13* and add the following code:

```
14
15 patch = {"email": "neo@thematrix.com"}
16 response = requests.patch('http://127.0.0.1:8000/
                            api/v1/addresses/<ID>/',
                            data=patch)
17 print(response.text)
```

Line 16 sends a `PATCH` request to our RESTful API to change the email for Neo's address.

Rerun the test script and check if the email was updated.

We're almost ready for testing; only two methods are left in the `AddressDetail` class to be tested: *retrieving* the details for a specific address and *deleting* an address. Now, try to add the required code for these methods to the test script yourself and test it.

For your reference or to compare, you can find the complete test script on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter05`.

With this, we finish the mixins-based views and continue with how generic class-based views work and how they are an improvement over mixins.

Creating a generic class-based view

Generic class-based views go another step further in simplicity because they combine the individual mixins classes. So, instead of having separate `ListModelMixin` and `CreateModelMixin` classes, we have a single `ListCreateAPIView` class covering both mixins. And the generic `RetrieveUpdateDestroyAPIView` class even combines three mixins:

1. `RetrieveModelMixin`
2. `UpdateModelMixin`
3. `DestroyModelMixin`

Furthermore, we only have to define a `queryset` object and a `serializer_class` object, and the generic class-based view does all the rest. This leads to more compact code, and we can overwrite our `views.py` file with this reduced code:

```
 1 from rest_framework import generics
 2 from .models import Address
 3 from .serializers import AddressSerializer
 4
 5 class AddressList(generics.ListCreateAPIView):
 6     queryset = Address.objects.all()
 7     serializer_class = AddressSerializer
 8
 9 class AddressDetail(generics.
                       RetrieveUpdateDestroyAPIView):
10     queryset = Address.objects.all()
11     serializer_class = AddressSerializer
```

To use generic class-based views, *line 1* imports the `generics` class from DRF.

Line 5 bases the `AddressList` class on the generic `ListCreateAPIView` class. And *lines 6-7* set the `queryset` and `serializer_class` objects.

Lines 9-11 do the same for the `AddressDetail` class, which is based on the `RetrieveUpdateDestroyAPIView` class.

With this, we reduced the views code by as much as 68 percent compared to mixins views. Now, let's test if our RESTful API still works as expected. To do so, run the tests from the `test_api.py` script and check the results. Comment out specific tests as desired to create different test scenarios or add tests if you wish.

Generic class-based views are a significant improvement as they require less code than `APIView` and mixins-based views. But we can do even better with `ModelViewSet`-based views, as we'll see in the following subsection.

Creating a ModelViewSet-based view

We've saved the best for last. Or at least the easiest, with `ModelViewSet`-based views, which combine *all* CRUD operations and reduce the required view code even more. To implement this view type, overwrite the `views.py` file with this code:

```
1 from rest_framework import viewsets
2 from rest_framework.permissions import AllowAny
3 from .models import Address
4 from .serializers import AddressSerializer
5
6 class AddressViewSet(viewsets.ModelViewSet):
```

```
7    queryset = Address.objects.all().order_by("name")
8    serializer_class = AddressSerializer
9    permission_classes = [AllowAny]
```

Technically, the `ModelViewSet`-based view only takes seven lines of code. But we've added permissions for our RESTful API through *lines 2* and *9*. We allow anybody, but we could limit access by setting `permission_classes` to `IsAuthenticated` or another permission class.

Lines 6-9 define the `AddressViewSet` class and are all we need to process all CRUD operations and corresponding HTTP methods from `GET` to `DELETE`.

`ModelViewSet`-based views simplify not only the view definition but also the definition of RESTful API endpoints in the app's `urls.py` file, which we can change to this content:

```
 1 from django.urls import include, path
 2 from rest_framework import routers
 3 from .views import AddressViewSet
 4
 5 ver = 'v1'
 6
 7 router = routers.DefaultRouter()
 8 router.register(f'api/{ver}/addresses', AddressViewSet)
 9
10 urlpatterns = [
11     path('', include(router.urls)),
12 ]
```

We applied the `routers` class in *lines 2*, *7*, and *8* to cover all CRUD operations. This opens up all the endpoints we need without defining them as paths in `urlpatterns`. *Line 8* does the crucial part and connects the base endpoint to the `AddressViewSet` class.

Finally, *line 11* includes all the default endpoints as paths through the `router` object. Now, rerun the tests to check if everything still works as required.

Okay – this ends our exploration of class-based views. We followed the evolution from `APIView`-based views to `ModelViewSet`-based views. And we saw how every evolution step was better than its predecessors, with the `ModelViewSet`-based view being the simplest and most powerful of all. Therefore, we'll keep the `ModelViewSet`-based view for our RESTful API.

Before DRF provided class-based views, we had to implement function-based views. Even though we prefer class-based views, we'll still look at how function-based views work in the next subsection because there might be rare cases where you need them.

Developing a function-based view

Together with `APIView` class-based views, function-based views require the most code to implement. However, function-based views appeal to some developers because they only use a function decorator and directly refer to HTTP methods, which makes them transparent and easy to follow.

Let's create function-based views for our RESTful API to see how they look and work. Start by renaming your existing `views.py` file `views_cbv.py` to save your ultimate **class-based view** (**CBV**).

Next, make a new `views.py` file and enter the following code:

```
 1 from rest_framework.decorators import api_view
 2 from rest_framework.response import Response
 3 from rest_framework import status
 4 from .models import Address
 5 from .serializers import AddressSerializer
 6
 7 @api_view(['GET', 'POST'])
 8 def address_list_view(request):
 9    if request.method == 'GET':
10      addresses = Address.objects.all()
11      serializer = AddressSerializer(addresses,many=True)
12      return Response(serializer.data)
13    elif request.method == 'POST':
14      serializer = AddressSerializer(data=request.data)
15      if serializer.is_valid():
16         serializer.save()
17         return Response(serializer.data,
                      status=status.HTTP_201_CREATED)
18      return Response(serializer.errors,
                      status=status.HTTP_400_BAD_REQUEST)
19
```

Line 1 imports the `api_view` decorator, and in *line 7*, we apply this decorator to set the `address_list_view` function in *line 8* as a function-based view and to handle `GET` and `POST` requests.

Furthermore, the `address_list_view` function in *line 8* receives the `request` parameter. The `if...elif` construction of *lines 9* and *13* queries the request method and executes the corresponding processing. If the request method is `GET`, then the view returns all addresses in *line 12*, and if the request method is `POST`, the view serializes the request data in *line 14* and adds a new address in *line 16*.

The explicit referrals to `GET` and `POST` make this code transparent and easy to follow. But it also makes the code comprehensive, especially if we add the code for handling individual `GET`, `PUT`, and `DELETE` requests:

```
20 @api_view(['GET', 'PUT', 'DELETE'])
21 def address_detail_view(request, pk):
22     try:
23         address = Address.objects.get(pk=pk)
24     except Address.DoesNotExist:
25         return Response(status=status.HTTP_404_NOT_FOUND)
26
27     if request.method == 'GET':
28         serializer = AddressSerializer(address)
29         return Response(serializer.data)
30     elif request.method == 'PUT':
31         serializer = AddressSerializer(address,
                                          data=request.data)
32         if serializer.is_valid():
33             serializer.save()
34             return Response(serializer.data)
35         return Response(serializer.errors,
                           status=status.HTTP_400_BAD_REQUEST)
36     elif request.method == 'DELETE':
37         address.delete()
38         return Response(status=status.HTTP_204_NO_CONTENT)
```

Line 20 decorates the `address_detail_view` function to handle individual `GET`, `PUT`, and `DELETE` requests, and the `address_detail_view` function receives the `pk` parameter in *line 21* to select a specific address.

Furthermore, for function-based views, we have to select an address explicitly as the `try...except` construction in *lines 22-25* does. Then, the `if...elif` construction in *lines 27-38* processes the request methods.

To make this view work, we also need to update the app's `urls.py` file. Comment out to present content and add this code:

```
1 from django.urls import path
2 from .views import address_list_view, address_detail_view
3
4 ver = 'v1'
5
6 urlpatterns = [
7   path(f'api/{ver}/addresses/', address_list_view),
8   path(f'api/{ver}/addresses/<int:pk>/',
```

```
          address_detail_view),
9 ]
```

Lines 7 and *8* connect the endpoints to the function-based views.

Altogether, this is quite some code and the most comprehensive code base of all view types. But it does the work, as you'll see when you repeat the tests for the function-based view version of the RESTful API.

When you're done testing, you have to undo the function-based view because we want to apply the `ModelViewSet`-based view for our RESTful API. To do so, take these steps:

1. Delete `views.py` or rename it if you want to keep an example of a function-based view for later reference.
2. Rename the `views_cbv.py` file `views.py` to restore the `ModelViewSet`-based view.
3. Remove the last additions for the function-based views in `urls.py` and uncomment the code for the `ModelViewSet`-based view.
4. Rerun the test script to check if our RESTful API works as expected.

Great – you've made it and learned the essentials of class- and function-based views, enabling you to build RESTful APIs with DRF. With our RESTful API fully running, let's see how we can browse a DRF RESTful API in the following subsection.

Browsing a DRF RESTful API

Browsing is an alternative and easy way to test a DRF RESTful API. In the *Setting up DRF* section, we added a path to the project's `urls.py` file to make our RESTful API browsable. Now that our model, serializer, views, and endpoints are ready, let's see how API browsing works. While your Django project still runs, open a browser and navigate to `http://127.0.0.1:8000/api/v1/addresses/`. DRF now shows a list of all the addresses you've entered. You see a form at the bottom of the page with the address class fields from the model and serializer. Follow these steps to test creating a new address through a `POST` request:

1. In the **Name** and **Address** field, enter the values of your choice.
2. In the **Postalcode** field, enter the value `MKL-9067831-DUNBOTZA-62300-LEFT`.
3. In the **City**, **Country**, and **Email** fields, enter the values of your choice.
4. Click **Post** the save the address.

What happens? You get an error message saying the postal code exceeds the maximum of 20 characters. Great – this failure works. Now, let's make this a happy path test:

1. In the **Postalcode** field, enter a valid value of your choice.
2. Click **Post** again to save the address.

Now, our RESTful API accepts the address and adds it to the `address` collection in MongoDB.

> **Important note**
>
> **Postman** is one of the best-known tools for testing APIs. If you have experience testing APIs with Postman, feel free to test our RESTful API with Postman.
>
> Otherwise, it might be interesting to explore Postman. You can use it for free at `https://www.postman.com/`.

Testing through forms is for our convenience, but in real life, clients send requests with JSON data to our RESTful API. Luckily, we can test this as well from our web browser with these steps:

1. In the form section, click **Raw data**.

   * *A request simulation form opens.*

2. Leave the **Media type** field at `application/json`.
3. In the **Content** field, enter values of your choice in the JSON message.
4. Click **POST** then save the address.

   * *The RESTful API adds the new address to the database.*

Click **Get** to show a list of all addresses. Okay – it's fair to say that creating and reading works for our RESTful API. But what about the other CRUD operations? Let's test updating and deleting with these steps:

1. Navigate to `http://127.0.0.1:8000/api/v1/addresses/1/` or any other valid ID.

   * *DRF shows the address instance page with options for* **Delete**, **Put**, *and* **Patch**.

2. If needed, click **Raw data** to show the request simulation form.
3. In the **Content** field, change multiple values from the JSON message.
4. Click **Put**.
5. Check if the updates came through.
6. Next, change one value in the JSON message, click **Patch**, and check the update.
7. Finally, click **Delete** and click **Delete** again to confirm.
8. From the menu, click **Address List** and ensure that our RESTful API has removed the address.

Excellent – all CRUD operations work for our RESTful API. With this, we conclude building RESTful APIs with DRF. However, there is one more aspect to cover: error handling, as we'll cover in the following section.

Error handling

Clients will make mistakes or send invalid data when sending requests, and our RESTful API needs to catch them and inform the client about what went wrong to enable the client to anticipate accordingly. The client-server communication for a DRF-based RESTful API in case of an error generally covers these steps:

1. The client sends a request to an endpoint.
2. The RESTful API at the endpoint parses the request.
3. The URL mechanism and the serializer in the RESTful API validate the request and catch an error.
4. The RESTful API sends an error message and a corresponding HTTP status code back to the client.
5. The client parses the request.
6. The client signals the error and catches it with alternate processing.

To implement the RESTful API part in this process, we need to handle possible errors, which falls apart in the following areas:

- Handling wrong formatted requests, such as a missing data object, when creating new data through a `POST` request
- Handling validation errors where the data object contains invalid data such as a negative item price

Let's look at how we can handle these error types for our RESTful API and start with bad requests.

Handling wrong-formatted requests

Depending on the type of class-based or function-based views we applied, the view logic handles bad requests itself, or we must write the error-handling code ourselves.

The `ModelViewSet` view of our RESTful API has built-in error handling for wrong-formatted requests. If we omit an ID in our request when we want to delete an address, we automatically receive a `405` (`Method not allowed`) response.

The same applies to mixins-based and generic class-based views. But for `APIView`-based views and function-based views, we need to write error-handling code ourselves. For example, to catch a bad request for the `POST` method in an `APIView`-based view, we need to write code like this:

```
1 class AddressList(APIView):
2     def post(self, request, format=None):
3         serializer = AddressSerializer(data=request.data)
4         if serializer.is_valid():
5             serializer.save()
6             return Response(serializer.data,
                      status=status.HTTP_201_CREATED)
```

```
7        return Response(serializer.errors,
                         status=status.HTTP_400_BAD_REQUEST)
```

The serializer plays a crucial role in error handling. First, *line 4* checks if the serialized data is valid. If so, then we return a positive response in *line 6*. But if the data is invalid, we return the errors the serializer encountered and status code `400` (`Bad request`).

This way of error handling covers errors such as missing elements in a data object or the request itself. But what if we also want to validate the provided data to ensure it's within a specific range or doesn't exceed a maximum value? Then, we need to catch validation errors, as we'll see in the following subsection.

Handling validation errors

If you have a RESTful API that processes new articles, you don't want the article price to be zero or negative. The most logical way to solve this is to apply Django's built-in validators in the model like this:

```
from django.db import models
from django.core.validators import MinValueValidator

class Product(models.Model):
   ...
   price = models.IntegerField(
       default=1
       validators = [MinValueValidator(1)]
   )
```

The validator ensures we get a response such as `"Ensure this value is greater than or equal to 1."` with HTTP status code `400` when we provide a price lower than `1`.

But what if our validation is more complex and requires the prices for products in product group `XYZ` to be higher than 125? The Django model can't handle this, but luckily, the serializer can with a code setup like this:

```
 1 from rest_framework import serializers
 2 from .models import Product
 3
 4 class ProductSerializer(serializers.ModelSerializer):
 5    class Meta:
 6        model = Product
 7        fields = ("id", "description", "group", "price"
 8
 9    def validate(self, data):
10       if data['group'] == 'XYZ' and data['price'] < 125:
11          raise serializers.ValidationError('The product
                    price for product group XYZ must 125
```

```
                    or higher ')
12       return data
```

Line 9 defines the `validate` method for the `ProductSerializer` class.

Line 10 checks if the price for a product from the `XYZ` group is less than 125. If so, *line 11* raises a custom error, which the RESTful API sends back to the client. If the price is correct, *line 12* returns the `data` object.

This way, we can check for any condition we want and send useful error information to the client if a request fails our validation.

> **Important note**
>
> If you've ever built a web application that consumes a RESTful API, you most likely know how vital clear error information is. So, think carefully about the error messages you send back from your RESTful API to help client developers. Be as specific as possible–for example, do not just say that an address is wrong but also return that it's longer than the maximum of *x* characters.

Now, back to our RESTful API. Can you write a serializer validation that ensures the postal code is seven characters long when the country is Utopia? Sure you can. Also, could you modify the test script to test both an invalid and a valid postal code for Utopia?

You can compare your serializer with an example version on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter05`.

Okay – this should be enough for you to handle request and validation errors gracefully. And with that, we have come to the end of creating RESTful APIs for Django microservices.

Summary

In this chapter, we learned how to build RESTful APIs with DRF as part of a Django microservices application. We started with exploring what RESTful APIs are before we moved over to creating such APIs with DRF. We ended with error handling to make DRF-based RESTful APIs resilient and to provide helpful feedback to clients in case of bad requests or invalid data.

With this, you know what RESTful APIs are and how to develop them with DRF. Furthermore, for the sample application, you have the RESTful API in place as the gateway for the microservices to the database.

In the next chapter, we'll build the microservices and learn how to orchestrate them with RabbitMQ and Celery.

6

Orchestrating Microservices with Celery and RabbitMQ

The producers and workers in the microservices architecture need a task queue mechanism to communicate. In most cases, this is one-way communication where a producer offloads a task to a worker. But workers can send back a response if your application requires such a scenario. We can choose from a range of task queue managers, but Celery and RabbitMQ are dominant in the Django universe, so we apply them in our microservices application.

In this chapter, you'll learn the foundation of task queues, including different task queue modes such as Publish-Subscribe and routing. Next, you'll learn about the most common Django task queue managers, Celery and RabbitMQ. Then, you'll learn how to build asynchronous tasks (microservices), including the response scenario. And you'll master how to orchestrate producer-worker communication with Celery and RabbitMQ. Finally, you'll learn how to monitor tasks and task queues.

By the end of this chapter, you'll know the principles of task queuing and how to develop and orchestrate Django microservices with Celery and RabbitMQ, including keeping track of tasks.

To cover this, this chapter addresses the following topics:

- Introducing task queues
- Exploring Celery and RabbitMQ
- Creating and running asynchronous tasks
- Monitoring tasks and task queues

This is the key chapter in developing microservices with Django. So, dig in, and before you know it, you'll have mastered the art of building Django microservices, significantly expanding your toolset as a Django developer.

Technical requirements

Our sample microservices application will use fuzzy string matching to match new addresses to existing base addresses to keep our `address` collection uncluttered. We'll use fuzzy string matching based on the *Levenshtein Distance metric*. This metric is a measure of the similarity between two strings and calculates the minimum number of operations needed to make two strings identical.

As you might expect, the Python community has already developed `pip` packages that provide this type of string matching. We'll apply the `rapidfuzz` package, which you install with this command in your virtual environment:

```
$ pip install rapidfuzz
```

In the *Creating and running asynchronous tasks* section, we'll look into the details and parameters for `rapidfuzz`, but if you want more information, check this URL: `https://github.com/rapidfuzz/RapidFuzz`.

Furthermore, Redis and RabbitMQ must be running for the microservices we'll be creating. You already installed and started them in *Chapter 3*, *Setting Up the Development and Runtime Environment*, but you might have closed them since. If needed, start Redis with this command:

```
$ docker run --name my-redis --network my-network -d -p
6379:6379 redis
```

And start RabbitMQ with this command:

```
$ docker run -it --rm --name my-rabbitmq
--network my-network -p 5672:5672 -p 15672:15672
rabbitmq:3.8-management-alpine
```

In this chapter, we will program a fair amount of code. You'll learn the most if you type the code yourself, but if you want, you can always copy code or parts of it from GitHub: `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter06`.

Introducing task queues

Producers and workers are independent and need a mechanism for producers to offload tasks to workers and, occasionally, for workers to send back a response to a producer. The microservices architecture uses task queues for this communication. A task queue is what its name implies: a queue where producers place tasks and workers pick up those tasks. In its most basic form, a task queue operates like this:

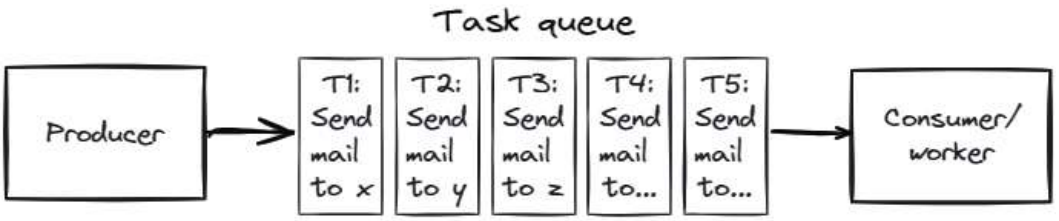

Figure 6.1 – A basic task queue

Besides this basic scenario, task queue managers support more complex scenarios such as the following:

- **Work queues**, where multiple workers share the tasks
- **Publish-Subscribe**, where multiple workers receive and process all tasks
- **Request-Response**, where workers send back a response to the producer

Let's look at how these scenarios work, including code examples based on the `pika` package for the producers and workers and with RabbitMQ as the task queue manager. We'll start with the work queue scenario.

Implementing the work queue scenario

In the work queue scenario, workers compete to execute the provided tasks, and each task is processed only once by one of the available workers. For example, two workers process incoming tasks for sending an email:

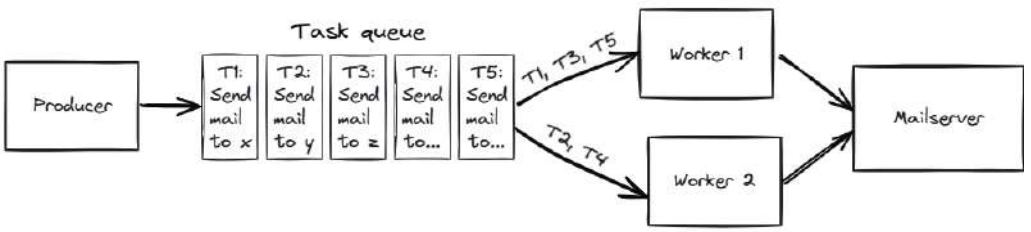

Figure 6.2 – The work queue scenario

Worker 1 processes tasks 1, 3, and 5, and *Worker 2* takes care of tasks 2 and 4. This way, each task is executed only once. The producer code could look like this:

```
1 import json
2 import pika
3
4 connection = pika.BlockingConnection(
```

```
                    pika.ConnectionParameters('localhost'))
 5 channel = connection.channel()
 6
 7 channel.queue_declare(queue='mail_queue', durable=True)
 8
 9 for idx in range(1, 6):
10    mail_task = {
11        'name': f'M. McDoe{idx}',
12        'email': f'mcdoe{idx}@yyz.com',
13        'subject': 'Our Django offer',
14        'body': 'Dear...'
15    }
16
17    channel.basic_publish(exchange='',
                            routing_key='mail_queue',
                            body=json.dumps(mail_task),
                            properties=pika.BasicProperties(
                            delivery_mode=
                               pika.DeliveryMode.Persistent))
18
19 connection.close()
```

Lines 4-7 set up a task queue named `mail_queue`.

To simulate multiple tasks, the `for` iteration in *line 9* creates five tasks, whereas *lines 10-15* construct the task, and *line 17* offloads the task to the queue.

If you want to follow along, take the following steps:

1. Inside the `django-microservices` directory, create a subdirectory called `tq-scenarios`.
2. Open the `tq-scenarios` directory in VS Code, create a new file called `wq_producer.py`, and enter or copy the producer code into this file.

Next, we need workers to process the email tasks. Their implementation looks like this:

```
 1 import json
 2 import pika
 3
 4 connection = pika.BlockingConnection(
                    pika.ConnectionParameters('localhost'))
 5 channel = connection.channel()
 6
 7 channel.queue_declare(queue='mail_queue', durable=True)
 8
 9 def callback(ch, method, properties, body):
```

```
10    mail_message = json.loads(body)
11    print(f"Email sent to {mail_message['name']}")
12
13 channel.basic_consume(
                queue='mail_queue',
                on_message_callback=callback,
                auto_ack=True)
14
15 channel.start_consuming()
```

Lines 4-7 connect the worker to a task queue named `mail_queue`.

Line 9 declares a `callback` function that executes whenever the worker picks up a task. Within this function, *line 10* parses the task, and *line 11* simulates sending an email.

Lines 13-15 make the worker listen for new tasks in the `mail_queue` task queue and execute the `callback` function when a new task is received.

To see the work queue scenario in action, take the following steps:

1. Save the worker code into a new file called `wq_worker.py`.
2. Open a terminal, move to the `tq-scenarios` directory, and run the worker script.
3. Open a second terminal and start the second worker here.
4. Open a third terminal and run the `wq_producer.py` script to offload the email tasks to the workers.

When you check the terminal of the first worker, you'll see that it processed tasks 1, 3, and 5. And in the terminal for the second worker, you'll notice that it executed tasks 2 and 4. So, our task queue manager, RabbitMQ, distributed the tasks among the workers, as the work queue scenario implies.

To stop the workers, type *Ctrl + C* in their corresponding terminal shells.

The work queue scenario suits situations where you want to distribute the task load over different workers. Depending on your application requirements, these workers can run on different servers for optimal load balancing.

In other cases, you might want all workers to process all tasks in a Publish-Subscribe scenario, which we'll explore next.

Implementing the Publish-Subscribe scenario

In the Publish-Subscribe scenario, all workers receive and process each task provided to the task queue. For example, we could have two workers that process the same task differently. *Worker 1* receives an email task and sends a corresponding email. *Worker 2* gets the same email task and writes a log item for the email to a logging file for later reference. Such a scenario looks like this:

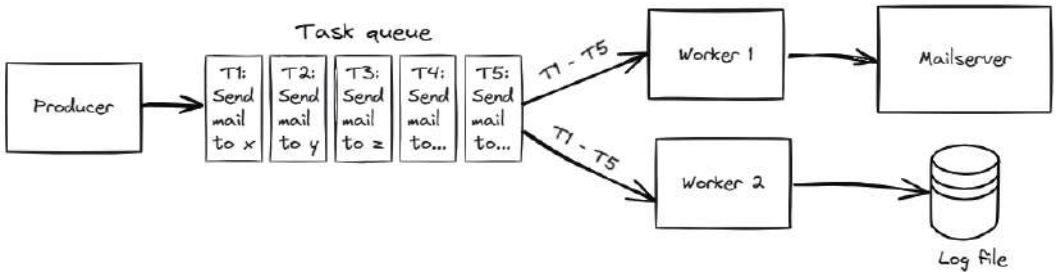

Figure 6.3 – The Publish-Subscribe scenario

To facilitate this scenario type, RabbitMQ applies the *exchange* mechanism. An exchange is a dispatcher that creates temporary task queues and *publishes* tasks from producers into the temporary task queues it manages. Then, workers *subscribe* to the same exchange to receive all tasks from the temporary task queues. Code-wise, the producer looks like this:

```
 1 import json
 2 import pika
 3
 4 connection = pika.BlockingConnection(
                   pika.ConnectionParameters('localhost'))
 5 channel = connection.channel()
 6
 7 channel.exchange_declare(exchange='email',
                           exchange_type='fanout')
 8
 9 for idx in range(1, 6):
10     mail_task = {
11         'name': f'M. McDoe{idx}',
12         'email': f'mcdoe{idx}@yyz.com',
13         'subject': 'Our Django offer',
14         'body': 'Dear...'
15     }
16
17     channel.basic_publish(exchange='email',
                             routing_key='',
                             body=json.dumps(mail_task))
```

```
18
19 connection.close()
```

Line 7 is the central part of the Publish-Subscribe scenario and sets up an exchange called `email`. The exchange's type is `fanout`, which directs the exchange to send (to fan out) all tasks to all workers.

Line 17 offloads the tasks to the `email` exchange.

Enter or copy this code in a file called `ps_producer.py` and create a file named `ps_worker_1.py` with the following code to send emails asynchronously:

```
 1 import json
 2 import pika
 3
 4 connection = pika.BlockingConnection(
                    pika.ConnectionParameters('localhost'))
 5 channel = connection.channel()
 6
 7 channel.exchange_declare(exchange='email',
                           exchange_type='fanout')
 8 result = channel.queue_declare(queue='', exclusive=True)
 9 queue_name = result.method.queue
10 channel.queue_bind(exchange='email', queue=queue_name)
11
12 def callback(ch, method, properties, body):
13    mail_message = json.loads(body)
14    print(f"Email sent to {mail_message['name']}")
15
16 channel.basic_consume(queue= queue_name,
                         on_message_callback=callback,
                         auto_ack=True)
17
18 channel.start_consuming()
```

Line 7 subscribes the worker to the `email` exchange, but since the exchange creates temporary queues as needed, the worker must determine the temporary queue.

To do so, *line 8* collects the temporary task queue, and *line 9* determines the name of the temporary queue. Then, *line 10* connects the worker to the temporary queue. Finally, *line 16* taps the tasks from the temporary task queue and executes the `callback` function for each task.

Next, create a file named `ps_worker_2.py` with the following code to write a log item for every email sent:

```
1 import datetime
2 import json
```

```
 3 import pika
 4
 5 connection = pika.BlockingConnection(
                  pika.ConnectionParameters('localhost'))
 6 channel = connection.channel()
 7
 8 channel.exchange_declare(exchange='email',
                           exchange_type='fanout')
 9 result = channel.queue_declare(queue='', exclusive=True)
10 queue_name = result.method.queue
11 channel.queue_bind(exchange='email', queue=queue_name)
12
13 def callback(ch, method, properties, body):
14    mail_message = json.loads(body)
15    now = datetime.datetime.now()
16    print(f"[{now.strftime('%Y-%m-%d %H:%M:%S')}] email
             sent to {mail_message['name']}")
17
18 channel.basic_consume(queue= queue_name,
                        on_message_callback=callback,
                        auto_ack=True)
19
20 channel.start_consuming()
```

This worker simulates writing a log item in *lines 15-16*. The remainder of the worker is the same as *Worker 1*.

To run this Publish-Subscribe scenario, take these steps:

1. Open a terminal and start the first worker.
2. Open a second terminal to start the second worker.
3. Open a third terminal and run the producer script.

In both worker terminals, you'll see that the corresponding worker processed tasks 1 to 5.

The Publish-Subscribe scenario suits situations where you want multiple workers to process the same task differently. A variant of the Publish-Subscribe scenario is the *routing* scenario, where workers subscribe to a specific subset of tasks. Check the RabbitMQ documentation at `https://www.rabbitmq.com/tutorials/tutorial-four-python.html` for an example of the routing scenario.

Okay –, this leaves us with one scenario: the Request-Response scenario, which we'll look into next.

Implementing the Request-Response scenario

In the Request-Response scenario, the producer offloads a task to a worker and waits for the worker's response. Essentially, this scenario breaks the rule that microservices are independent because the producer waits for a response, which creates a dependency.

Nevertheless, you might encounter application requirements where a Request-Response scenario is a good option – for example, to offload a heavy task to a powerful server in a situation where the waiting time for the producer isn't an issue. From your experience, you might know calculation or data engineering programs that run for hours on a plain server and only take a few minutes on a multi-core server with a vast amount of internal memory.

In such cases, a Request-Response scenario is a suitable solution and can look like this:

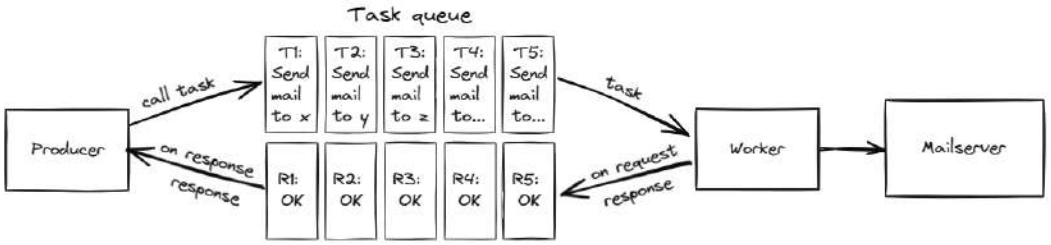

Figure 6.4 – The Request-Response scenario

The critical part of the Request-Response scenario lies in the producer because it needs a mechanism to wait for and fetch the response. A class is an easy way to send a task request to a worker and process the corresponding response. In the following subsection, we'll look at a class implementation for a Request-Response producer.

Creating the producer

Let's start with the requesting producer that offloads tasks to a responding worker to send an email and waits for the worker's response. To implement the producer, create a file called `rr_producer.py` and enter the following code into it:

```
1 import json
2 import pika
3 import uuid
4
5 class Mailer(object):
6   def __init__(self):
7       self.connection = pika.BlockingConnection(pika.
                    ConnectionParameters(host='localhost'))
8       self.channel = self.connection.channel()
9       result = self.channel.queue_declare(
```

```
10                                  queue='', exclusive=True)
11      self.callback_queue = result.method.queue
12      self.channel.basic_consume(
                 queue=self.callback_queue,
                 on_message_callback=self.on_response,
                 auto_ack=True)
13      self.response = None
14      self.corr_id = None
15
16  def on_response(self, ch, method, props, body):
17     if self.corr_id == props.correlation_id:
18        self.response = body
19
20  def call(self, mail_task):
21     self.response = None
22     self.corr_id = str(uuid.uuid4())
23     self.channel.basic_publish(
            exchange='',
            routing_key='mail_queue',
            properties=pika.BasicProperties(
                reply_to=self.callback_queue,
                correlation_id=self.corr_id,
            ),
            body=json.dumps(mail_task))
24     self.connection.process_data_events(
                                       time_limit=None)
25     return self.response
26
27 mail_task = {
28   'name': f'M. McDoe',
29   'email': f'mcdoe@yyz.com',
30   'subject': 'Our Django offer',
31    'body': 'Dear...'
31 }
32
33 mailer = Mailer()
34 response = mailer.call(mail_task)
35 print(response.decode('utf-8'))
```

Quite a lot is happening in the producer, which all revolves around the `Mailer` class declared in *line 5*. Let's dissect the `Mailer` class according to its main parts:

- The `__init__` method in *line 6*.

  The first part of the `Mailer` class is the `__init__` method in *line 6*, which initializes the connection to RabbitMQ. *Lines 7-10* are standard practice. But *lines 11* and *12* are specific for the Request-Response scenario because they set up a callback queue for responses and state that the `on_response` method should be executed when a message (response) arrives in the callback

  queue.

  The `__init__` method closes with the `response` and `corr_id` variables. `corr_id` holds the correlation ID that uniquely connects a task and its corresponding response.

- The `on_response` method in *line 16*.

  The second part is the `on_response` method in *line 16*. This method sets the response when it's received from the worker, and the correlation IDs of the task and the response match.

- The `call` method in *line 20*.

  The third and final part of the `Mailer` class is the `call` method in *line 20*. This is the entry point for offloading a task to a microservice worker. *Line 22* creates a **Universally Unique Identifier** (**UUID**) for the correlation ID. Then, *line 23* sets the `mail_queue` task queue with these `pika` properties:

1. `reply_to` for assigning the callback queue where the worker should drop the response.
2. `correlation_id` to exchange the generated UUID to the worker.

   Finally, the `call` method sets the time limit for the response in *line 24* to endless, and *line 25* returns the actual response from the worker.

   The remainder of the producer starts with setting the task in *lines 27-31*. Then, *line 33* creates a `Mailer` object. Next, *line 34* calls the `call` method with the task as the parameter and the `response` variable to receive and hold the response. Finally, *line 35* decodes the byte string response and prints it out.

Let's summarize the producer's processing to be sure we understand its flow:

1. The process starts with initializing the task queue and a callback queue for the response.
2. Then, the `call` method offloads the task to the worker, including information about the callback queue where the worker must drop its response and the correlation ID to match a task and its corresponding response.
3. Finally, the `on_response` method fetches the response from the worker.

Okay, so far, for the producer. Now, let's look at the worker.

Creating the worker

The worker executes the task to send an email on request and returns a response when finished. To simulate a real-world situation, the worker randomly responds with `OK` and `NOK` (**Not OK**). To implement the worker, create a file named `rr_worker.py` with the following code:

```
 1 import json
 2 import pika
 3 import random
 4
 5 connection = pika.BlockingConnection(
              pika.ConnectionParameters(host='localhost'))
 6 channel = connection.channel()
 7 channel.queue_declare(queue='mail_queue', durable=True)
 8
 9 def on_request(ch, method, props, body):
10     mail_message = json.loads(body)
11     print(f"Email sent to {mail_message['name']}")
12     response = "OK" if random.choice([True, False]) ==
                  True else "NOK"
13     ch.basic_publish(exchange='',
                        routing_key=props.reply_to,
                        properties=pika.BasicProperties(
                        correlation_id=props.correlation_id),
                        body=response)
14     ch.basic_ack(delivery_tag=method.delivery_tag)
15
16 channel.basic_qos(prefetch_count=1)
17 channel.basic_consume(queue='mail_queue',
                         on_message_callback=on_request)
18 channel.start_consuming()
```

Compared to the producer, the worker implementation is less comprehensive. So, there is no need for a class, and we can suffice with an `on_request` function, which executes the task and compiles the response.

Lines 5-7 connect the worker to RabbitMQ.

Line 9 defines the `on_request` function, where *line 11* emulates sending an email, and *line 12* randomly sets the response to `OK` or `NOK`. *Line 13* connects the worker to the callback queue, where it must drop its response, including the correlation ID that matches the response and the originating task. The `on_request` function ends with *line 14*, which sets the acknowledgment for the received task messages.

Finally, *line 16* sets the **quality of service** (**QoS**) for the communication between producer and worker, and *line 17* connects the worker to the `mail_queue` task queue and sets the `on_request` function as the callback function to execute when tasks arrive.

Let's summarize once again to cover all the steps for the producer and the worker:

1. *Producer*: Initialize the task queue and a callback queue for the response.
2. *Producer*: Execute the `call` method to request task execution by the worker.
3. *Worker*: Fetch and execute the task via the `on_request` function.
4. *Worker*: Compile the response and place it in the callback queue.
5. *Producer*: Fetch the response from the callback queue.
6. *Producer*: Execute the `on_response` method to return the response.
7. *Producer*: Process the response.

That was quite some code and explanation. Well done! It's high time to see the Request-Response scenario in action. To do so, take these steps:

1. Open a terminal and start the worker.
2. Open a second terminal and run the producer script.
3. Repeat the producer script several times.

In the worker terminal, you see the print output that emulates sending email. In the producer terminal, you alternately see `OK` and `NOK` as the received response.

Excellent; with this, we finished introducing task queues, where we learned how to implement the work queue and the Publish-Subscribe and Request-Response scenarios for communication between producers and workers.

While learning about task queues and communication scenarios, we took RabbitMQ as our task queue manager for granted. But RabbitMQ and its alternative, Celery, determine the way we implement and code producers and workers. Therefore, we take a closer look at them in the next subsection.

Exploring Celery and RabbitMQ

Celery and RabbitMQ are the most common task queue managers in the Django microservices universe. They are both open source products, and Celery only runs on Linux and macOS. Technically, Celery is a task queue manager, and RabbitMQ is a message queue manager or broker. But for Django microservices, we both use Celery and RabbitMQ for task queue management, and therefore we address them as task queue managers.

In *Chapter 2*, *Introducing the Django Microservices Architecture*, we looked into Celery and RabbitMQ. But as you might jump straight into here, let's see what they offer. We'll start with Celery because it's most used in Django microservices applications and the easiest way to orchestrate microservices.

Celery

Celery is an open source task queue system written in Python and available as a Python package (`https://pypi.org/project/celery/`). Celery runs on top of message queue brokers such as RabbitMQ and Redis, enabling producers to offload tasks asynchronously to workers.

Celery brings the tasks for both the producer and the worker together in a single `tasks.py` file, which leads to a Django setup like this:

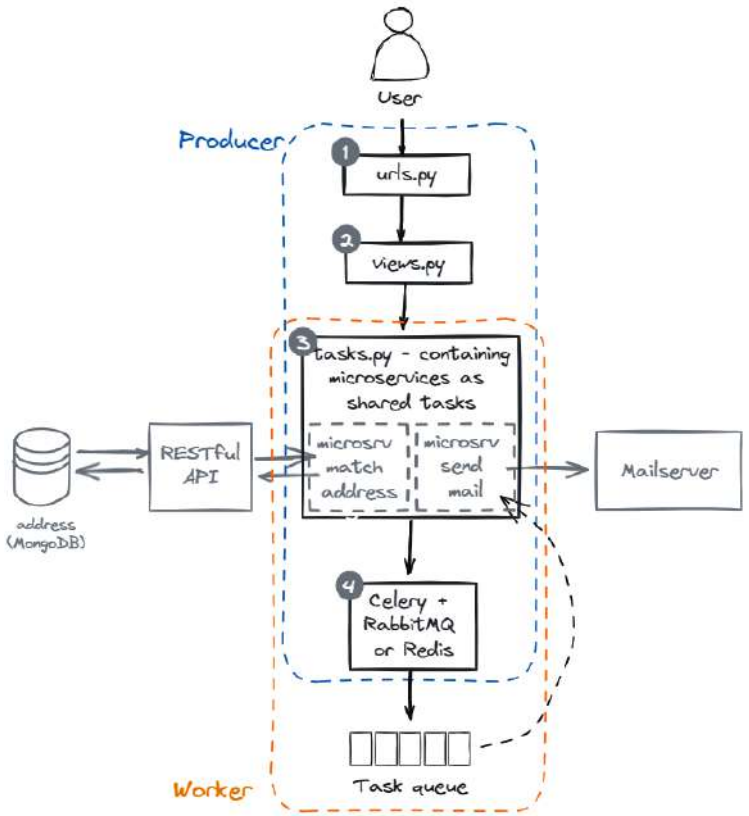

Figure 6.5 – Celery-Django setup with overlapping producer and worker

Let's walk through this setup from a Django perspective:

1. *Producer*: A user calls for a web page action via the `urls.py` file.
2. *Producer*: The `urls.py` file passes the action to the `views.py` file.
3. *Producer*: The `views.py` file links to the `tasks.py` file and offloads a task to a shared task microservice via Celery.

   Okay, so far, for the producer; now, the worker gets into action.

4. *Worker*: The shared task microservice in the `tasks.py` file notices the task and executes its functionality.

What's unique about Celery is the producer and worker overlap in the `tasks.py` file. At runtime, the producer and the worker are separate processes, but their code definition is the same as a shared task in the `tasks.py` file. In the *Creating and running asynchronous tasks* section, we'll look at what shared Celery tasks actually look like and how Celery operates. But first, we'll look at RabbitMQ from a Django perspective.

RabbitMQ

RabbitMQ is an open source message queue broker based on the **Advanced Message Queuing Protocol** (**AMQP**), and we apply RabbitMQ from Django through the `pika` Python package. With RabbitMQ, we have a separate producer and worker, both in implementation and at runtime. In fact, a RabbitMQ worker operates as a Django extension from a `scripts` directory in a setup like this:

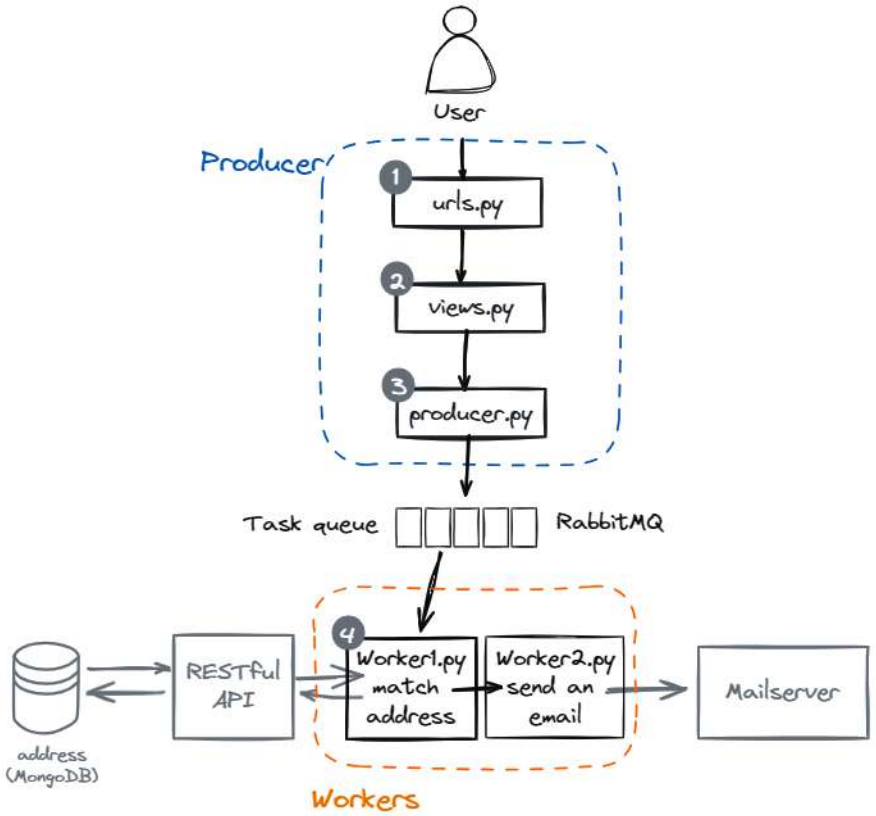

Figure 6.6 – RabbitMQ-Django setup with separate producer and workers

If we explore this setup from a Django perspective, we distinguish these steps:

1. *Producer*: A user calls for a web page action via the `urls.py` file.
2. *Producer*: The `urls.py` file passes the action to the `views.py` file.
3. *Producer*: The `views.py` file links to the `producer.py` file and offloads a task to a worker via the RabbitMQ task queue.

   Okay, so far, for the producer; now, the worker gets into action.

4. *Worker*: The worker, which runs as a Django extension, notices the task and executes its functionality.

Schematically, the RabbitMQ setup looks simpler than the Celery setup. However, implementation-wise, the Celery setup is more straightforward to realize, which is why Celery is so prevalent in Django microservices applications. So, apply Celery if you can and apply RabbitMQ if you must.

To cover and illustrate both, we'll build asynchronous tasks with Celery and RabbitMQ in the following section. Due to its popularity, we'll start with Celery and then build the same task with `pika` and RabbitMQ. This way, you get a good insight into their pros and cons so that you can decide when to apply one or the other.

Creating and running asynchronous tasks

We'll continue the development of our Django microservices application by building the actual microservices for matching addresses and sending emails, which will run as asynchronous tasks called by the Django subscription app. This is where the Django app and microservices sit in the overview of the application:

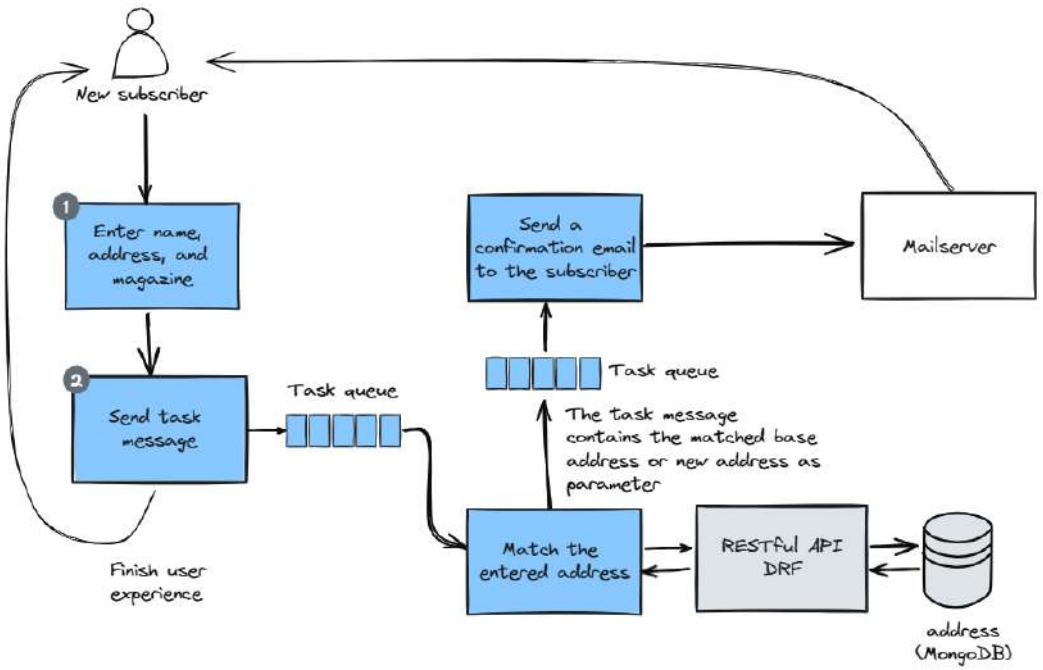

Figure 6.7 – The position of the app and microservices in the sample application

The Django subscription app will display a form where users subscribe and enter their addresses. When the user finishes entering, the app offloads a task to match the entered address to existing base addresses to a microservice. This microservice utilizes our RESTful API to retrieve all addresses and add a new address, and it offloads the task of sending a confirmation email to another microservice.

> **Important note**
>
> In our sample application, one microservice will offload a task to another microservice. If we apply Celery, we could use the mechanism of *task chaining* as an alternative to one microservice calling another. Check the Celery documentation at `https://docs.celeryq.dev/en/stable/userguide/canvas.html#chains` if you want to see how this works.

Because Celery is the most common task queue solution for Django applications, we'll build the Celery-based version of the microservices first. Next, we'll develop the RabbitMQ-based version for more insight into how task queuing works in comparison.

Creating and running a Celery-based task

Celery works with function decorators to define a Python function as an asynchronous task. The most common Celery decorators are `@app.task` and `@shared_task`. Basically, the decorators do the same, but `@app.task` is more suitable in a one-off Django project, and you apply `@shared_task` in reusable or shared projects.

We'll build our sample application with shared tasks, but first, we'll develop a quick `@app.task` producer and worker to see how these work.

Building a @app.task-based producer and worker

For simplicity, we'll build a producer and worker apart from Django. The worker will write a log item to a collection in our `Subscription` database on MongoDB. Since we're working apart from Django, we'll use the `pymongo` package to write to MongoDB directly.

We'll apply RabbitMQ as the underlying message queue broker for Celery, so be sure RabbitMQ is running.

First, we'll create a new collection for the log items. Move to the MongoDB web interface and create a collection called `subscription_logitem` inside the `Subscription` database.

Next, we'll build the worker that writes log items to the collection. To do so, move into the `django-microservices` directory and make a new `logwriter` directory. And inside the `logwriter` directory, create a file named `celery_worker.py` with this code:

```
import datetime
from celery import Celery
from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi

app = Celery('log', broker='pyamqp://guest@localhost//')

@app.task()
```

```
 9 def write_logitem(application, logmessage):
10    now = datetime.datetime.now()
11    uri = "mongodb+srv://django-microservice:
           <password>@<cluster>/?
           retryWrites=true&w=majority"
12    client = MongoClient(uri, server_api=ServerApi('1'))
13    subscription_db = client["Subscription"]
14    logitem_col = subscription_db["subscription_logitem"]
15
16    logitem_col.insert_one({'time': now.strftime(
                                '%Y-%m-%d %H:%M:%S'),
                              'app': application,
                              'logmessage': logmessage})
17    print(f"[{now.strftime('%Y-%m-%d %H:%M:%S')}]
           - new logmessage entered")
```

Line 6 sets RabbitMQ as the message broker for Celery through the `broker` parameter and connects the worker to the `log` task queue.

Line 8 decorates the `write_logitem` function in *line 9*, which assigns the function as an asynchronous Celery-driven task.

Lines 11-14 open the connection to our MongoDB database and the `subscription_logitem` collection. Replace `<password>` and `<cluster>` with your values in *line 11*.

Finally, *line 16* writes a log item to the collection, and *line 17* prints a message to the terminal.

Okay – the worker is ready. Now, let's build the producer, which only requires two lines of code. Create a file named `celery_producer.py` with this code:

```
1 from celery_worker import write_logitem
2
3 write_logitem('Subscription',
                'New subscription entered for J. Watson')
```

Line 1 imports the asynchronous `write_logitem` task from the worker. Here, we see the overlap between a Celery producer and worker in action as they apply the same task.

Line 3 offloads the task to write a log item by calling the `write_logitem` task with the required parameters for the application and the log message.

Okay – we're done building and are ready to test the log writer microservice with these steps:

1. Open a terminal and start the worker with this command:

```
$ celery -A celery_worker worker -l info
```

2. Open a second terminal and run the producer script.

Move over to the MongoDB web interface and open the `subscription_logitem` collection, or refresh the page if you have the collection open to check the new log item. Cool, isn't it? The producer just offloads the task, and the worker does the actual work asynchronously. Celery also supports synchronized task execution, as we'll see in the following subsection.

Creating a Request-Response variant with Celery

In the *Implementing the Request-Response scenario* subsection, we learned how to build such a scenario with `pika` and RabbitMQ. Celery also supports this scenario with just a simple function return. To implement this, take these steps:

1. Press *Ctrl* + *C* to stop the Celery worker.
2. Change *line 17* in `celery_worker.py` to this code:

```
...
17   Return 'Log message entered'
```

3. Change *line 3* in `celery_producer` to the following:

```
...
3 response = write_logitem('Subscription', 'New
             subscription entered for I. Adler')
```

4. Add this line to `celery_producer` to print the response:

```
Print('Received response:', response)
```

5. Start the Celery worker again, run the producer, and check the response output.

Important note

Because the Request-Response scenario is so easy with Celery, it might be tempting to implement it to check if tasks execute correctly. However, the Request-Response scenario creates a dependency between the producer and worker, so we lose the benefits of microservices. Therefore, it's better to avoid this scenario whenever possible and implement error handling and signaling for the microservices, as we'll explore in *Chapter 11*, *Best Practices for Microservices*.

This finishes the `@app.task` producer and worker. We've built them apart from Django, but you can easily integrate them into Django, as we'll see in the following subsection, where we'll develop the Django app and the `@shared_task`-based microservices for our sample application.

Building the Django app and the @shared_task-based microservices

Okay – now, it's time to pick up the development of our sample application again. Next, we'll build the Django app with the subscription form and the Celery microservices that match an entered address with existing base addresses and send a confirmation email.

Let's start with the Django app, which will have the following form where new subscribers enter their addresses:

Figure 6.8 – The address entry form

And the directory structure and key files for the app look like this:

```
subscription_celery/
|
├── subscription/
| ├── templates/subscription
| | ├── base.html
| | ├── subscription.html
| | └── success.html
| |
| ├── forms.py
| ├── models.py
| ├── tasks.py
| ├── urls.py
| └── views.py
|
├── subscription_celery/
| ├── __init__.py
| ├── celery.py
| ├── urls.py
| └── settings.py
|
└── manage.py
```

When applying Celery shared tasks, we place the shared tasks in a `tasks.py` file in the app directory, which Celery automatically processes. This does require that we initialize Celery as an app in a `celery.py` file in the lower project directory and that we define the Celery app in a `__init__.py` file in the lower project directory.

> **Important note**
>
> Our Django project has one app with one `tasks.py` file for the Celery workers. But when a project has multiple apps, each app can have its own `tasks.py` file, and Celery automatically discovers all shared tasks from all task files.

We'll create a `tasks.py` file, a `celery.py` file, and a `__init__.py` file in a moment, but first, we'll create and set up a Celery-based Django project and app with these steps:

1. Inside the `django_microservices` directory, create a Django project called `subscription_celery`.
2. Move into the `subscription_celery` directory and create a Django app named `subscription`.

3. Open the `settings.py` file in VS Code and add the subscription app to `INSTALLED_APPS`:

```
INSTALLED_APPS = [
  ...
  'subscription',
]
```

4. Next, update the `DATABASES` definition to the following:

```
DATABASES = {
  'default': {
     'ENGINE': 'djongo',
     'NAME': 'Subscription',
     'ENFORCE_SCHEMA': False,
     'CLIENT': {
             'host': 'mongodb+srv://django-
              microservice:<password>@<cluster>'
        }
    }
}
```

   Replace `<password>` and `<cluster>` with your values.

5. Then, add the console as the backend for email delivery:

```
# Email setting
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

6. Finally, set Redis as the message queue broker for Celery:

```
# Celery settings
CELERY_BROKER_URL = "redis://localhost:6379"
CELERY_RESULT_BACKEND = "redis://localhost:6379"
```

   `CELERY_BROKER_URL` sets Redis as the message broker, and `CELERY_RESULT_BACKEND` sets Redis as the storage for Celery results.

7. Save and close `settings.py`.

Okay – this sets the project and app. Our next task is to create an app and subscription form.

Creating the app and form

The Django app provides a form where new subscribers enter their addresses. This app and form are similar to the app and form from *Chapter 4*, *Cloud-Native Data Processing with MongoDB*. So, we'll only present the code here without explanation.

To create the app and the form, make a directory called `templates/subscription` inside the `subscription_celery/`

`subscription` directory and create the following files:

- A `subscription_celery/subscription/templates/`
- `subscription/base.html` file:

```
{% load static %}
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible"
        content="IE=edge">
   <meta name="viewport" content="width=device-width,
              initial-scale=1.0">
    <link rel="stylesheet"
               href="https://unpkg.com/bamboo.css">
    <title>Subscription</title>
</head>
<body>
    {% block content %}
    {% endblock content %}
    <small>Celery driven</small>
</body>
</html>
```

- A `subscription_celery/subscription/templates/`
- `subscription/subscription.html` file:

```
{% extends 'subscription/base.html' %}
{% block content %}
<h2>Enter the address where we can deliver your magazine</h2>

<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Subscribe" />
</form>
{% endblock content %}
```

- A `subscription_celery/subscription/templates/`

- `subscription/success.html` file:

```
{% extends 'subscription/base.html' %}
{% block content %}
<h1>Thanks!</h1>
<p>You'll receive the latest edition of our magazine within
three days.</p>
{% endblock content %}
```

- A `subscription_celery/subscription/forms.py` file:

```
from django import forms

class SubscriptionForm(forms.Form):
    name = forms.CharField(label="Your name")
    address = forms.CharField(label="Address")
    postalcode = forms.CharField(label="Postal code")
    city = forms.CharField(label="City")
    country = forms.CharField(label="Country")
    email = forms.EmailField(label="Email")
```

- A `subscription_celery/subscription/models.py` file:

```
from djongo import models

class Address(models.Model):
    _id = models.ObjectIdField()
    name = models.CharField(max_length=120)
    address = models.CharField(max_length=120)
    postalcode = models.CharField(max_length=20)
    city = models.CharField(max_length=120)
    country = models.CharField(max_length=6)
    email = models.EmailField()
```

- A `subscription_celery/subscription/urls.py` file:

```
from django.urls import path
from subscription.views import SubscriptionFormView, SuccessView

app_name = "subscription"

urlpatterns = [
    path("subscription/",
             SubscriptionFormView.as_view(),
             name="subscription"),
    path("success/", SuccessView.as_view(),
```

```
                name="success"),
]
```

Important note

Technically, we don't need an implementation for `models.py` because we use the RESTful API to retrieve and update addresses. But still, we implement it to be able to migrate our model to the MongoDB database if needed.

To complete the app and form, we'll also need to define a `views.py` file, but because this file imports and uses the shared tasks from the `tasks.py` file, we'll create shared tasks files first in the following subsection.

Creating the shared tasks files

Now, we'll create a `tasks.py` file that holds the shared tasks for matching an address and sending a confirmation email. Sending an email is straightforward, but matching an address needs some explanation.

To avoid our `address` collection getting cluttered with variants of the same address due to typos, capitalization, and different spelling, we'll use fuzzy string matching to check if our collection holds a base address for a newly entered address.

If a new address matches an existing base address, the new address gets the address of the (highest) matching base address. Otherwise, we store the new address with its provided address.

To match new addresses, we'll apply the `rapidfuzz` Python package, which implements the Levenshtein Distance metric. This metric calculates a matching score between `0` and `100` for two strings. `0` means no matching, and `100` means the strings are identical. For example, the matching score for the `East River Street 14` base address and an entered variant such as `East river str. 14` is `89`.

Depending on the application and data requirements, we can set a minimum for the score to match. A minimum score of 70 is a reasonable starting point, and in that case, we would have a match for `East river str. 14` and would store the new address with the `East River Street 14` base address.

The `rapidfuzz` package facilitates multiple matching methods. We'll use the `ratio` method. If you're interested in other methods, check the `rapidfuzz` documentation at `https://rapidfuzz.github.io/RapidFuzz/Usage/fuzz.html`.

Important note

Technically, the `ratio` method calculates the *Indel distance*, a variant of the Levenshtein Distance metric. But since Levenshtein is more common, we speak of this metric.

All right – let's create microservices (shared tasks) for matching an address and sending an email. The matching microservice will utilize the RESTful API we created in *Chapter 5*, *Creating RESTful APIs for Microservices*.

Inside the `subscription` directory, create a file named `tasks.py` with the following code:

```
 1 import requests
 2 from celery import shared_task
 3 from django.core.mail import send_mail
 4 from rapidfuzz import fuzz
 5 from subscription.models import Address
 6
 7 @shared_task
 8 def match_address_task(address):
 9   response = requests.get(
              'http://127.0.0.1:7000/api/v1/addresses/')
10   addresses = [a_address['address'] for a_address in
                                        response.json()]
11  top_score = 0
12  min_score = 70
13  match_address = address["address"]
14  for base_address in addresses:
15    score = round(fuzz.ratio(address["address"].lower(),
             str(base_address).lower()))
16    if score >= top_score and score >= min_score:
17       top_score = score
18       match_address = base_address
19    if top_score == 100:
20       continue
21    print(f'Match address: {match_address} > Score:
          {top_score}')
22    address = {"name": address["name"],
23               "address": match_address,
24               "postalcode": address["postalcode"],
25               "city": address["city"],
26               "country": address["country"],
27               "email": address["email"]
28               }
29    response = requests.post('http://127.0.0.1:7000/
                 api/v1/addresses/', data=address)
30    print(f"New address inserted for {address['name']}")
31
32    send_email_task.delay(address["name"],
                            address["email"])
```

```
33
34 @shared_task
35 def send_email_task(name, email):
36     send_mail(
         "Your subscription",
         f"Dear {name},\n\nThanks for subscribing to our
         magazine! You'll receive the latest edition
         of our magazine within three days.\n\nCM
         Publishers",
         magazine@cm-publishers.com,
         [email],
         fail_silently=False,
     )
```

Let's dissect the microservices in this file and start with the matching task. *Line 7* decorates the `match_address_task` function in *line 8* so that the function becomes a Celery shared task.

The function starts by reading all known base addresses from the MongoDB collection into a Python list. *Line 9* uses the `requests` package to retrieve all addresses through our RESTful API. *Line 10* transforms the response into a list named `addresses`, the base input for address matching.

The matching begins with some initialization in *lines 11-13*. The `top_score` variable will hold the overall highest match score. The `min_score` variable sets our minimum score for a successful match. `match_address` initially holds the entered address (street) but will be overwritten by the street from a base address if we have a successful match.

The actual matching takes place in *lines 14-21*. *Line 14* loops over all retrieved base addresses, and *line 15* calculates per base address the matching score for the entered address. If the computed score is higher than the minimum score and the top score so far in *line 16*, *line 17* preserves the new top score, and *line 18* preserves the street from the current base address. So, at the end of looping over the base addresses, `match_address` will hold the street of the highest matching base address or the originally entered street if there's no match at all.

Line 19 checks if `top_score` has reached the maximum of 100 to avoid unnecessary matching. If so, we have a perfect match and can stop matching via the `continue` statement in *line 20*.

At this point, we have determined the correct value for the street, so *lines 22-28* create an `address` object that *line 29* writes to the `address` collection via a `POST` request to our RESTful API.

Finally, the matching task offloads sending an email to the `send_email_task` function, which is defined as a decorated shared task in *lines 34-35*.

Okay – this finishes the microservices and sets our Celery workers. We're almost ready; we only need to complete the Django app and prepare Celery to discover the microservices (shared tasks). The Django app needs a `views.py` file to handle the form input and to offload the address-matching task to a Celery worker. To do so, we enter the following code into `views.py`:

```
 1 from django.views.generic.base import TemplateView
 2 from django.views.generic.edit import FormView
 3 from subscription.forms import SubscriptionForm
 4 from subscription.tasks import match_address_task
 5
 6 class SubscriptionFormView(FormView):
 7    template_name = "subscription/subscription.html"
 8    form_class = SubscriptionForm
 9    success_url = "/success/"
10
11    def form_valid(self, form):
12       task_message = {
13          "name": form.cleaned_data["name"],
14          "address": form.cleaned_data["address"],
15          "postalcode": form.cleaned_data["postalcode"],
16          "city": form.cleaned_data["city"],
17          "country": form.cleaned_data["country"],
18          "email": form.cleaned_data["email"]
19       }
20       match_address_task.delay(task_message)
21
22       return super().form_valid(form)
23
24 class SuccessView(TemplateView):
25    template_name = "subscription/success.html"
```

Line 4 imports the `match_address_task` shared task from the `tasks.py` file. As a result, the `views.py` file becomes the *producer* in our Celery microservices scenario.

The `FormView` class in *line 6* and the `TemplateView` class in *line 24* are standard practice, as you know, for Django. *Lines 7-8* handle the address entry form, and *line 9* sets the page to show when a subscriber successfully enters an address.

Line 11 defines the `form_valid` function, which executes when a subscriber successfully enters an address. The function creates a `task_message` object, which holds the entered address in *lines 12-19* and then offloads the matching task in *line 20*.

Excellent! There is only one thing left. We need to prepare for Celery to discover our shared tasks. We'll need two files for this. First, create a file named `celery.py` inside the lower `subscription_celery` directory with this code:

```
1 import os
2 from celery import Celery
3
4 os.environ.setdefault("DJANGO_SETTINGS_MODULE",
                       "subscription_celery.settings")
5 app = Celery("subscription_celery")
6 app.config_from_object("django.conf:settings",
                         namespace="CELERY")
7 app.autodiscover_tasks()
```

Celery needs this file for its settings and to run as an app that discovers shared tasks. To do so, *lines 4* and *6* give Celery access to its settings via the `CELERY` namespace in the Django `settings.py` file.

Furthermore, *line 5* creates the app, and *line 7* makes the app discover our shared tasks.

Finally, we need to enable the Celery shared tasks to run as independent workers. We do this by creating or updating the `__init__.py` file in the lower `subscription_celery` directory with this code:

```
1 from .celery import app as celery_app
2
3 __all__ = ("celery_app",)
```

Line 1 imports the Celery app from the `celery.py` file. *Line 3* enables the Celery app, including the Celery shared tasks from `tasks.py`, to be started as a worker through the `celery` command.

All right – that's it, and you have made it. Well done! Let's experience the satisfaction of seeing how our sample application works. We need a few terminals for this:

1. In the first terminal, start our RESTful API from its directory with this command:

   ```
   $ python3 manage.py runserver 7000
   ```

   We run the RESTful API via port `7000` to set it apart from the Django subscription app, which will run through the standard Django port `8000`.

2. In a second terminal, move into the upper `subscription_celery` directory and execute this command to start the shared tasks as Celery workers:

   ```
   $ celery -A subscription_celery worker --loglevel=info
   ```

This starts the Celery app and shared tasks through the `__init__.py` and `celery.py` files. It discovers the shared tasks from `tasks.py`, as you see in the terminal output:

```
[tasks]
  . subscription.tasks.match_address_task
  . subscription.tasks.send_email_task
```

3. In a third terminal, start the Django subscription app.

Okay – all parts of our sample application are running. Let's test the application with these steps:

1. Open your browser and navigate to `http://127.0.0.1:8000/subscription/`.
2. Enter a complete address and click **Subscribe**.
3. Check the `address` collection in MongoDB for the new address. A great and satisfying feeling that it's there, isn't it?
4. Go back to your browser, go back to the subscription page, and enter an address with a street slightly different from the first address.
5. Click **Subscribe** again.
6. Check the matching score in the terminal where the Celery worker runs.
7. Check MongoDB to see how the application entered the second address.
8. Test the application further with lower-scoring addresses and an address with no matching at all.
9. Close the terminals when you're done.

Fantastic! You've built a Celery-based Django microservices application, including a Django producer app and Celery workers. This significantly expands your skills as a Django developer, so it's time to update your resume. But before you do that, explore how to build the same functionality with `pika` and RabbitMQ. This empowers you even more as a Django developer because you'll master different techniques for building Django microservices.

Creating and running a RabbitMQ-based task

With RabbitMQ and `pika`, we'll have separate files for each microservice that stand apart from the Django app. Still, the microservices must run within the Django context to utilize Django functionality. Therefore, we must build the microservices as Django extensions that run within the Django context.

For this, we need the `django_extensions` package and a `scripts` directory where we store the Python scripts that act as extensions and contain our workers (microservices).

The Django subscription app for the RabbitMQ-based sample application only slightly differs from the Celery-based version. For instance, the form and the model are the same. So, we'll focus on the parts that differ in the following directory structure:

```
subscription_rabbitmq/
|
├── scripts/
| ├── email_worker.py
| ├── match_worker.py
|
├── subscription/
| ├── templates/subscription
| | ├── base.html
| | ├── subscription.html
| | └── success.html
| |
| ├── forms.py
| ├── models.py
| ├── producer.py
| ├── urls.py
| └── views.py
|
├── subscription_rabbitmq/
| ├── urls.py
| └── settings.py
|
└── manage.py
```

You'll find all the files for the RabbitMQ sample application on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter06/subscription_rabbitmq`.

Let's look at the worker files first and start with the email worker (`email_worker.py`):

```
 1 import json
 2 import pika
 3 from django.core.mail import send_mail
 4
 5 connection = pika.BlockingConnection(
 6          pika.ConnectionParameters(host='localhost'))
 7 channel = connection.channel()
 8 channel.queue_declare(queue='mail_queue', durable=True)
 9
10 def callback(ch, method, properties, body):
```

```
11    task_message = json.loads(body)
12    send_mail(
        "Your subscription",
        f"Dear {task_message['name']},\n\nThanks for
         subscribing to our magazine! You'll receive the
         latest edition of our magazine within three
         days.\n\nCM Publishers",
        magazine@cm-publishers.com,
        [task_message['email']],
        fail_silently=False,
      )
13    ch.basic_ack(delivery_tag=method.delivery_tag)
14
15 def run():
16    print("Waiting for email requests")
17    channel.basic_qos(prefetch_count=1)
18    channel.basic_consume(queue='mail_queue',
                on_message_callback=callback)
19    channel.start_consuming()
```

Lines 1-13 are familiar as we covered them in earlier examples, such as in the *Implementing the work queue scenario* subsection.

The `run` function, starting at *line 15*, is most new to us and ensures that the worker listens to the `mail_queue` task queue and runs endlessly until we stop it with *Ctrl + C*.

Next, we'll explore the matching worker (`match_worker.py`), most of which will look familiar to us:

```
 1 import json
 2 import pika
 3 import requests
 4 from rapidfuzz import fuzz
 5
 6 connection = pika.BlockingConnection(
 7 pika.ConnectionParameters(host='localhost'))
 8 channel = connection.channel()
 9
10 channel.queue_declare(queue='match_queue', durable=True)
11 channel.queue_declare(queue='mail_queue', durable=True)
12
13 def callback(ch, method, properties, body):
14    task_message = json.loads(body)
15    response = requests.get('http://127.0.0.1:7000/
                api/v1/addresses/')
16    addresses = [a_address['address'] for a_address in
```

```
                  response.json()]
    top_score = 0
    min_score = 70
    match_address = task_message["address"]
    for base_address in addresses:
        score = round(fuzz.ratio(
                      task_message["address"].lower(),
                      str(base_address).lower()))
        if score >= top_score and score >= min_score:
           top_score = score
           match_address = base_address
        if top_score == 100:
           continue

    print(f'Match address: {match_address}
           > Score: {top_score}')

    address = {"name": task_message["name"],
               "address": match_address,
               "postalcode": task_message["postalcode"],
               "city": task_message["city"],
               "country": task_message["country"],
               "email": task_message["email"]
               }
    response = requests.post('http://127.0.0.1:7000/
               api/v1/addresses/', data=address)
    print("New subscription added for",
           task_message["name"])

    channel.basic_publish(exchange='',
                      routing_key='mail_queue',
                      body=json.dumps(task_message),
                      properties=pika.BasicProperties(
                      delivery_mode=
                      pika.spec.PERSISTENT_DELIVERY_MODE)
                      )
    ch.basic_ack(delivery_tag=method.delivery_tag)

def run():
    print("Waiting for match address requests")
    channel.basic_qos(prefetch_count=1)
    channel.basic_consume(queue='match_queue',
```

```
48     on_message_callback=callback)
49    channel.start_consuming()
```

Lines 1-8 initialize `pika` and RabbitMQ.

Lines 10 and *11* define two task queues because the match worker needs to listen to the `match_queue` task queue for new tasks to execute, and it needs to offload tasks to the `mail_queue` task queue.

The `callback` function that starts in *line 13* contains the matching we saw earlier in the *Building the Django app and the @shared_task-based microservices* subsection, supplemented by offloading the email task in *lines 41* and *42*.

All right – the workers are ready. Now, let's look at the producer and the `views.py` file, which ties the producer and, indirectly, the workers to the Django app. This is the `producer.py` file:

```
1 import json
2 import pika
3
4 connection = pika.BlockingConnection(
              pika.ConnectionParameters(host='localhost'))
5 channel = connection.channel()
6 channel.queue_declare(queue='match_queue', durable=True)
7
8 def match_address_task_message(task_message):
9    channel.basic_publish(exchange='',
                       routing_key='match_queue',
                       body=json.dumps(task_message),
                       properties=pika.BasicProperties(
                       delivery_mode=
                       pika.spec.PERSISTENT_DELIVERY_MODE)
                       )
```

The producer is rather simple because it only defines the `match_address_task_message` function in *line 8*, which offloads a matching task to the `match_queue` task queue.

Consequently, the `views.py` file imports the producer, handles the subscription form, and calls the `match_address_task_message` function to offload address matching:

```
1 from django.views.generic.base import TemplateView
2 from django.views.generic.edit import FormView
3 from .forms import SubscriptionForm
4 from .producer import match_address_task_message
5
6 class SubscriptionFormView(FormView):
7     template_name = "subscription/subscription.html"
8    form_class = SubscriptionForm
```

```
 9    success_url = "/success/"
10
11    def form_valid(self, form):
12        task_message = {
            "name": form.cleaned_data["name"],
            "address": form.cleaned_data["address"],
            "postalcode": form.cleaned_data["postalcode"],
            "city": form.cleaned_data["city"],
            "country": form.cleaned_data["country"],
            "email": form.cleaned_data["email"]
        }
13        match_address_task_message(task_message)
14
15        return super().form_valid(form)
16
17 class SuccessView(TemplateView):
18     template_name = "subscription/success.html"
```

Line 4 imports the `match_address_task_message` function, and *line 13* calls this function when Django has validated the form subscription input.

To run the RabbitMQ version of the sample application, follow these steps:

1. Create a Django project named `subcription_rabbitmq` with a Django app called `subscription`.
2. Download the files from GitHub (`https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter06/subscription_rabbitmq`) into the appropriate directories.
3. Start the RESTful API from its directory with the following command:

   ```
   $ python3 manage.py runserver 7000
   ```

4. Open a second terminal, move into the upper `subscription_rabbitmq` directory, and execute this command to start the email worker as a Django extension:

   ```
   $ python3 manage.py runscript email_worker
   ```

   The `runscript` parameter starts the `email_worker.py` file as a Django extension. We don't need to provide the `.py` file extension because `runscript` automatically handles this.

5. Open a third terminal, and execute this command to start the match worker:

   ```
   $ python3 manage.py runscript match_worker
   ```

6. And open a fourth terminal to run the Django app.

Okay – let's test the RabbitMQ application with these steps:

1. Open your browser and navigate to `http://127.0.0.1:8000/subscription/`.

   * *You see the RabbitMQ-driven version of the address page.*

2. Enter a completely new address and click **Subscribe**.
3. Check the `address` collection in MongoDB for the new address.
4. Go back to your browser, go back to the subscription page, and enter an address with a street slightly different from the first address.
5. Click **Subscribe** again.
6. Check the matching score in the terminal where the matching worker runs.
7. Check MongoDB to see how the application entered the last entered address.
8. Close the terminals when you're done.

Excellent; you now also know how to build RabbitMQ-based Django microservices and can choose the best solution to fulfill your stakeholders requirements. You're now competent in orchestrating Django microservices with Celery and RabbitMQ.

Regarding microservice execution, we can inspect the terminals for our sample applications to check the status of microservices and tasks. But in production, we need to monitor tasks, as we'll explore in the following subsection.

Monitoring tasks and task queues

Monitoring is a field in itself that can stretch from simply checking task execution to statistical analysis for load balancing. In general, load balancing is the domain of system administrators and specialists. We, as developers, are primarily interested in task execution. RabbitMQ has built-in functionality for this, and for Celery, we'll apply the Flower package.

As our focus is on Celery, we'll start with Flower and then look into RabbitMQ.

Monitoring Celery tasks with Flower

The Flower Python package provides a web interface to monitor Celery tasks. Flower taps into running Celery workers and exposes their runtime information through a web interface. To see how Flower works, let's monitor our sample Celery application with the following steps:

1. Start the RESTful API from its directory:

```
$ python3 manage.py runserver 7000
```

2. Start the Celery workers:

```
$ celery -A subscription_celery worker --loglevel=info
```

3. Open a third terminal and start Flower with this command to tap into the Celery workers:

```
$ celery -A subscription_celery flower
```

4. Start the Django subscription app.
5. Open your browser, navigate to the subscription page, and enter an address.
6. Open a new tab in your browser and navigate to `http://127.0.0.1:5555/` to open the Flower interface.

   Flower opens by showing the workers for your workstation:

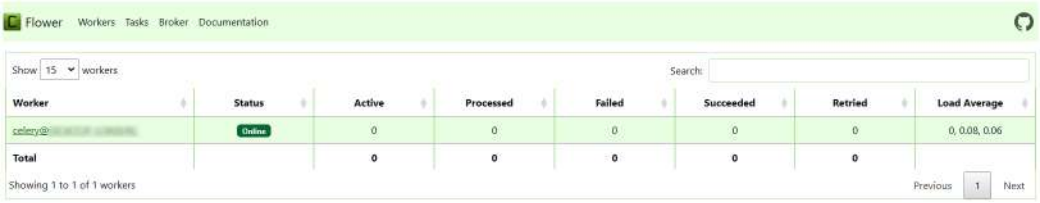

Figure 6.9 – Flower interface

7. Click **Tasks** to inspect the task details. You now see rows for the email and the match task. Inside the rows, the **args** column shows the arguments that the Django app passed to the tasks, which can help debug a microservice. Furthermore, the **Runtime** column shows the execution time for the task, which can help to trace bottlenecks.
8. Browse the different Flower interface options to get familiar with them.
9. Click **Documentation** to check the Flower documentation for further information. The documentation is also available at `https://flower.readthedocs.io/en/latest/`.

When ready, let's look at monitoring RabbitMQ.

Monitoring RabbitMQ tasks

RabbitMQ has a built-in web interface for monitoring and works differently from Celery and Flower because RabbitMQ only monitors the task queues and the message delivery. To check RabbitMQ monitoring, let's monitor our sample RabbitMQ application:

1. If needed, start the RESTful API from its directory:

```
$ python3 manage.py runserver 7000
```

2. Start the email and match worker with the following commands in separate terminals:

```
$ python3 manage.py runscript email_worker
$ python3 manage.py runscript match_worker
```

3. Open a fourth terminal to run the Django app.
4. Open your browser, navigate to the subscription page, and enter an address.
5. Open a new tab in your browser and navigate to `http://127.0.0.1:15672/` to open the RabbitMQ interface.
6. Log in with `guest` for both the **Username** field and the **Password** field.

Browse the interface to explore the options. Check the RabbitMQ documentation at `https://www.rabbitmq.com/monitoring.html` if you want more information on monitoring.

We only looked at the surface of task monitoring because monitoring is a field in itself and, therefore, too extensive to cover in full. But this gives you a good impression of the possibilities of monitoring and directions for diving deeper if you need to. And with this, we finish this chapter.

It's time to sit back and let it sink in that you have mastered developing Django microservices. This opens new areas for you as a Django developer and enables you to develop highly sophisticated Django applications.

Summary

This chapter covered the heart of developing Django microservices and started by introducing task queues and different task queue scenarios such as work queues and Publish-Subscribe.

Next, we looked at Celery and RabbitMQ, the most common task queue managers and brokers for Django applications. Both are open source tools but have different approaches for offloading tasks to microservices.

From there, we created and tested Celery- and RabbitMQ-based asynchronous tasks or microservices. This double approach gave us a good insight into the specifics of Celery and RabbitMQ so that we can select the best tool to meet application requirements.

Finally, we looked into task monitoring from a developer perspective.

With this, we have mastered the development and orchestration of Django microservices. In the next chapter, we'll learn how to test our microservices.

7
Testing Microservices

Testing is trying out software in different scenarios to ensure that our application meets the functional and quality requirements set by the stakeholders. In most software development teams, specialized software testers test the software for bugs and aspects such as performance or vulnerability.

Still, we developers also need to test our software before we hand it over to testing specialists, and in general, we use a division into white- and black-box testing for this purpose. We, developers, know the inside of the software as an open or white box and compile tests based on that technical knowledge. On the other hand, software testers see the software as a black box because they don't know what's happening inside and test software from a functional perspective.

A good approach for testing microservices by developers is testing the individual parts and trying out the application as a whole. That way, we'll cover the technical implementation of microservices and the functionality of the entire application, which, if successful, will give us an application good enough to hand over to the test specialists.

In this chapter, you'll learn about unit testing for the individual parts and end-to-end testing for the whole application.

By the end of the chapter, you'll know how to apply test (design) techniques such as boundary and equivalence testing for unit tests and how to set up an end-to-end test based on the user stories for our application. Moreover, you'll know how to automate your tests, as encouraged by agile software development.

To achieve this, this chapter covers the following topics:

- Introducing testing microservices
- Unit testing microservices
- End-to-end testing microservices
- Automated testing with Selenium

Testing is essential because it ensures we deliver proper software to take pride in. So, follow along to build the best software we can.

Technical requirements

We'll automate the testing of our sample application from automatically entering a new subscription until checking MongoDB for the corresponding new `address` document.

To automate the web page handling, we'll apply the `selenium` package and integrate its functionality into Python test scripts to cover all application parts. You install the `selenium` package with this command:

```
$ pip install selenium
```

Furthermore, you'll find the code samples for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter07`.

Introducing testing microservices

We test our software to ensure it fulfills the requirements set by the product owner and stakeholders. Software testing is a field in itself with different philosophies. To take the extremes, some say testers should know as little as possible about the software to test and test it in an exploratory way as a new user would. In contrast, others argue that testers should know as much as possible about the software to test so that testers can design test scenarios upfront and then test the software according to these scenarios.

We, developers, belong to the latter group because we know the software's inner and technical details, enabling us to perform white-box testing where we test scenarios based on our technical knowledge of the software under test.

For testing in general, and therefore also for testing microservices, we have to deal with these factors when we set up our tests:

- You can't test everything and all scenarios, so you must make choices. For example, if a field can hold a minimum value of 0 and a maximum value of 999999, it's impossible to test all values from 0 to 999999 because that would take too much time.
- Test as early as possible to reduce repair costs because the later you discover an error, the more expensive it is to fix it. For example, if you find a task specification error in a microservice just after developing, the fix only involves the microservice with little impact. But if you only discover the error at the end, you need to repair all involved producers, which takes a lot of time and has a high impact.
- Start with (isolated) parts and then move to components and the system as a whole because it reduces the time to find the cause of an error when you find it in an isolated part. Nevertheless, some errors only show up at the system level, so we also need to cover the whole system.

Considering these factors, we'll apply unit and end-to-end testing for our sample application. **Unit testing** will cover the individual microservices and enable us to catch errors as early as possible. **End-to-end testing** will cover the complete system and allow us to catch errors such as a wrong task message specification that we won't find with unit testing.

Due to its nature, software isn't tangible, making it sometimes difficult to imagine how testing elaborates on software. A comparison to testing something tangible, such as a car, can help.

When building a car, we'll start with parts such as the motor and the accelerator. When we finish the first version of the motor, we'll unit test it with some temporary cabling, fuel line, or battery. We'll do the same for the accelerator, and when we notice something wrong, we can quickly isolate the cause as we only need to check the part.

When all unit tests pass, we'll install the motor, the accelerator, and other parts until we assemble the car. Next, we'll test the car end to end, and then we might find that depressing the accelerator doesn't increase the motor speed. Both the motor and the accelerator work fine, but the connection between them (the *interface* in software terms) wasn't correctly implemented, which appeared to be the problem.

Okay – back to testing our sample application. We'll start with unit testing the microservices and the RESTful API. Then, we'll test the complete application end to end, which gives us this test coverage:

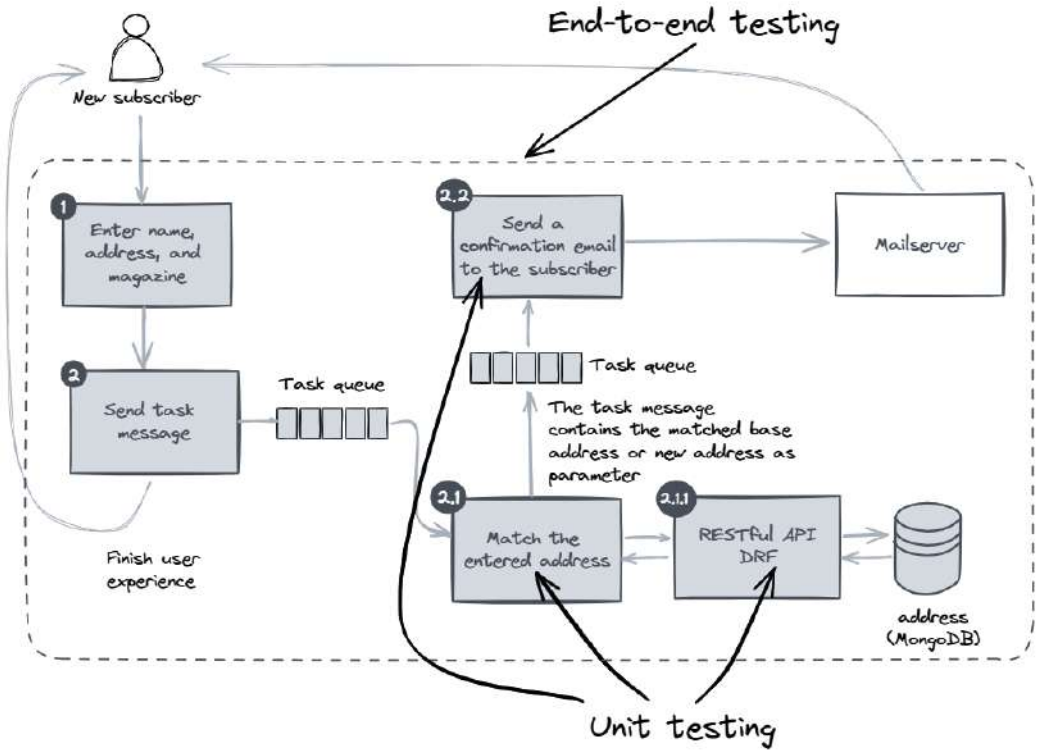

Figure 7.1 – Test coverage (unit and end-to-end testing)

For unit testing, we'll apply the Django test facilities and the standard `unittest` package because these are built in, simple, and effective.

For the end-to-end testing, we'll include the Django subscription app and start with creating and executing a manual test scenario. Next, we'll use Selenium to automate the test scenario.

We design and execute our tests for the Celery version of the sample application. Let's start with unit testing in the following section.

Unit testing microservices

Unit testing is trying out a software system's individual parts (units) to ensure it meets the requirements of the product owner and stakeholders. Unit tests are the domain of us developers, and we'll run them as soon as we have a stable version of a software component to discover errors as early as possible.

You can even apply the principles of **test-driven development** (**TDD**), which dictates writing unit tests first and then developing the software component until it satisfies the unit tests.

Whatever approach we choose, we prefer to automate unit tests so that we can repeat them quickly with every change, thus ensuring the quality of our software. If you or your team apply **continuous integration and continuous deployment** (**CI/CD**), you can integrate the automated unit tests in the deployment cycle to continuously deliver tested software.

To create unit tests with proper coverage, we need techniques for designing unit tests, and of the techniques available, we choose the following:

- **Happy path tests** that test a successful flow, such as entering a correct address
- **Boundary tests** that test for minimum and maximum values because errors appear to gather around boundaries

> **Important note**
>
> **Designing tests** is a field in itself, with several techniques and strategies. We apply some of the available techniques in this chapter, but if you want more information on test design, check a book such as *Software Test Design* by Simon Amey.

It's good to start with the happy path tests and then execute tests that specifically look for errors because if the happy path fails, we need to fix that first. Otherwise, we scrutinize an already buggy software component, and that's useless. So, let's begin with happy path tests for our microservices and RESTful API.

Creating and running happy path tests

Happy path tests are just what their naming implies: tests that cover common scenarios that we expect to succeed.

The purpose of happy path tests is to ensure common scenarios for our software work correctly as expected.

We base happy path tests on the requirements, and *use cases* are a great source to extract requirements from. Our product owner drew up this use case for entering a subscription address:

Overview	Enter a subscription address with these fields and restrictions: • `Name`, max length = `120` • `Address`, max length = `120` • `Postalcode`, max length = `15` • `City`, max length = `120` • `Country`, max length = `80` • `Email`, correct email format
Actor	A form in a Django app.
Trigger	A new subscriber enters an address.
Steps	1. Open the form. 2. Enter the address. 3. Validate the address. 4. Match the address. 5. Add the address to the database.
Outcome	*Success*: • The address is added to the database • The subscriber receives a confirmation email *Failure*: • Display an error message

Table 7.1 – Subscription address use case

This use case also involves microservices. When testing a chain of microservices, including a RESTful API, it's a good idea to start at the end of the chain and work forward because it helps isolate errors. For example, suppose the final RESTful API works fine, and you catch an error when you call the API from a preceding microservice. In that case, it's more likely to find the error in the microservice and sensible to focus on the microservice.

So, we start with a happy path test for the RESTful API. We implemented the restrictions of the use case in the model for our RESTful API like this in `/django-microservices/subscription_apis/address_api/`

`models.py`:

```
from django.db import models
class Address(models.Model):
    name = models.CharField(max_length=120, blank=True)
    address = models.CharField(max_length=120, blank=True)
    postalcode = models.CharField(max_length=20,
                                  blank=True)
    city = models.CharField(max_length=120, blank=True)
    country = models.CharField(max_length=80, blank=True)
    email = models.EmailField(blank=True)
```

The `max_length` arguments restrict the accepted field lengths, and the `EmailField` model assures the correct email format.

With the use case and the model in mind, we create a happy path test that should add an address to the `address` collection. And we'll use the Django test facilities and the standard `unittest` package for this. To follow along, open the `/django-microservices/subscription_apis/address_api/tests.py` file in VS Code and enter this test code:

```
 1 from rest_framework import status
 2 from rest_framework.test import APITestCase
 3 from .models import Address
 4
 5 class AddressTests(APITestCase):
 6       url = 'http://127.0.0.1:7000/api/v1/addresses/'
 7    def test_create_address(self):
 8       data = {"name": "A. Anderson",
                 "address": "Down Under 1",
                 "postalcode": "ZPT 1",
                 "city": "Gotham",
                 "country": "Northland",
                 "email": anderson@upside.com
                }
 9       response = self.client.post(self.url, data,
```

```
                    format='json')
10        self.assertEqual(response.status_code,
                           status.HTTP_201_CREATED)
11        self.assertEqual(Address.objects.count(), 1)
12        self.assertEqual(Address.objects.get().name,
                           'A. Anderson')
```

We build our RESTful API with DRF, and DRF also provides test facilities, which we import in *lines 1-2*.

In *line 3*, we also import the `Address` class from the model to check for the added address. Although Django and DRF use the model for testing, they don't create an address in the actual database and collection but create a temporary test database (`test_Subscription`) instead. When we run the tests, Django creates a `temp` database and deletes it when the tests finish.

Lines 5-7 define the happy path test by setting an `APITestCase`-based class with the `test_create_address` method.

Within the `test` method, *line 6* sets the endpoint for the RESTful API. *Line 8* creates the address (`data` object), and *line 9* calls the API and fetches the response.

With this, we finish the test execution. Next, we need to check if the response and result are as expected. In testing, we use assertions to check results, whereby we make assertions such as the value of this field needs to be equal to this value.

Likewise, the `test` method applies the `assertEqual` method, and *line 10* checks if the status code equals `201 – Created`. *Line 11* checks if the `address` collection holds one document as expected, and *line 12* checks if that one document holds the added name.

If all the assertions pass, the happy path test succeeds. If one or more assertions fail, the test fails. Either way, Django deletes the `temp` database when the test finishes.

Let's see this test in action with the following steps:

1. Start the RESTful API from its directory:

   ```
   $ python3 manage.py runserver 7000
   ```

2. Open another terminal, move into the `subscription_apis` directory, and run the test with this command:

   ```
   python3 manage.py test address_api -v 2
   ```

   The `test` argument makes `manage.py` run the `tests.py` file from the `address_api` app, as specified by the test app argument. The `–v 2` argument sets the test output to be verbose and show detailed test information.

See the test running automatically and inspect the test output. The test should pass `OK` and end with destroying the test database.

> **Important note**
>
> If you get an access error, make sure the MongoDB user has the `Atlas admin` role in MongoDB via Database Access | Edit | Built-in Role.

Testing the Celery workers

Now, let's create happy path unit tests for the Celery match and email microservices. These tests differ from the one for the RESTful API because we now test the microservices, which implicitly call the RESTful API. As a result, Django doesn't apply the temporary test database mechanism, and we need to look up the added address ourselves for the assertions and delete it afterward to prevent the database from being cluttered.

Next, we need to pause the test execution for a moment between matching (and adding) an address and looking up and deleting it because the microservices run asynchronously. So, we can't assume that the microservice creates an address instantly.

Furthermore, the `tests.py` file for the Celery subscription app must become the producer to offload the tasks to the microservices for testing. To implement this test setup, open the `/django-microservices/subscription_celery/`

`subscription/tests.py` file and enter this test code:

```
 1 import unittest
 2 from pymongo.mongo_client import MongoClient
 3 from pymongo.server_api import ServerApi
 4 from time import sleep
 5 from .tasks import match_address_task, send_email_task
 6
 7 class MatchTest(unittest.TestCase):
 8     last_id = 0
 9
10     def setUp(self):
11         con = "mongodb+srv://django-microservice:
               <password>@<cluster>/
               ?retryWrites=true&w=majority"
12         client = MongoClient(con,
                                server_api=ServerApi('1'))
13         db = client["Subscription"]
14         self.col = db["address_api_address"]
15         for last in self.col.find().sort('id',
                       -1).limit(1):
```

```
16            self.__class__.last_id = last['id']
17
18    def test_create_address(self):
19        task_message = {"name": "A. Antovic",
                          "address": "Down Under 1",
                          "postalcode": "ZPT 17",
                          "city": "Gotham",
                          "country": "Northland",
                          "email": antovic@upside.com
                         }
20        match_address_task.delay(task_message)
21        sleep(2)
22        for last in self.col.find().sort('id',
                    -1).limit(1):
23            self.assertEqual(last['name'], 'A. Antovic')
24            self.col.delete_one({'id': last['id']})
25
26 class MailTest(unittest.TestCase):
27    def test_send_email(self):
28        send_email_task.delay("D. Dhozos",
                                "dhozos@upside.com")
```

Line 1 imports the `unittest` package. *Lines 2-3* import `pymongo` to look up and delete an address. *Line 5* imports the shared tasks (microservices) so that `tests.py` becomes the producer.

Line 7 defines the test case for the matching worker. *Line 8* initializes the `last_id` class variable, which will hold the `id` value for the last entered address.

Lines 10-16 define the `setUp` method for the tests, which opens a connection to the MongoDB database and the `address` collection. *Lines 15-16* look up the `id` value of the last address and store this in the class variable.

Lines 18-20 offload the matching task. *Line 21* makes the test wait 2 seconds so that the match worker should be ready. *Line 22* looks up the added address, and *line 23* checks if the name of that address is equal to the one we provided. *Line 24* ends the test case for the match worker and deletes the added address. This way, we can run this repeatedly without cluttering the database.

Although the match worker sends a task to the email worker and implicitly tests the mail worker, we also explicitly test the mail worker in *lines 26-28*. We don't have an automated mechanism to assert the email sent, so we check the Celery worker's terminal output manually.

Okay – it's time for action. Run these tests with the following steps:

1. If needed, start the RESTful API:

```
$ python3 manage.py runserver 7000
```

2. If needed, start Redis or RabbitMQ as a message broker for Celery.
3. Start the Celery workers:

```
$ celery -A subscription_celery worker -l info
```

4. Open another terminal, move into the `/django-microservices/`
5. `subscription_celery` directory, and run the tests with this command:

```
python3 manage.py test subscription -v 2
```

Check the terminal output to ensure the tests ran successfully and the email was sent. Also, check the MongoDB web interface to ensure the test deleted the added address.

Okay – we covered creating a new address. Now, let's go a step further and test the matching mechanism like this:

1. Add a unique address that becomes a base address.
2. Add a new address with a street that diverts slightly from the base address from *step 1*.
3. Check with an assertion that the last added address has the same street as the base address.
4. Delete the added base address and variant.

Now, are you in for a challenge? Of course you are. So, you're challenged to build and add the tests for the matching mechanism, which cover *steps 1* to *4*.

For your reference, you'll find a sample of the test script on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter07`.

Great – our happy path unit tests are ready and running successfully. Now, let's see if boundary testing reveals errors in the next subsection.

Creating and running boundary tests

Boundary tests focus on the minimum and maximum (boundaries) of field values and lengths to ensure that all specified values process correctly. Values only apply to numerical fields, and length applies to text fields.

The reason for boundary testing is that errors tend to creep up around minimum and maximum values. For example, in some programming languages, an integer variable can hold a minimum of `-32768` and a maximum of `32767`. When an application form allows a lower or larger value to be entered, the underlying code can crash.

So, it's essential to test boundaries, and again, we use the requirements (of the *Overview* part) from the use case to design our tests:

Overview	Enter a subscription address with these fields and restrictions: • `Name`, max length = `120` • `Address`, max length = `120` • `Postalcode`, max length = `15` • `City`, max length = `120` • `Country`, max length = `80` • `Email`, correct email format

Table 7.2 – Boundary requirements

Ultimately, we need to test all fields, but to illustrate the principle of boundary testing, we'll focus on the `postalcode` field. We start with boundary testing the RESTful API and then test from the matching microservice.

For the RESTful API, we'll create these tests:

- *Scenario*: Maximum length
- *Value*: Postal code of 15 characters
- *Expected result*: Address normally accepted and processed
- *Scenario*: Too long
- *Value*: Postal code of 16 characters.
- *Expected result*: Address rejected with an error message

To implement these tests, open the `/django-microservices/`

`subscription_apis/address_api/tests.py` file again and add these tests to the `AddressTests` class:

```
29 ...
30    def test_create_address_postalcode_max(self):
31        data = {"name": "B. Botir",
```

```
32                "address": "Down Under 1",
33                "postalcode": "ABCDEFGHIJ12345",
34                "city": "Gotham",
35                "country": "Northland",
36                "email": botir@upside.com
37                }
38        response = self.client.post(self.url, data,
                                      format='json')
39        self.assertEqual(response.status_code,
                         status.HTTP_201_CREATED)
40        self.assertEqual(Address.objects.count(), 1)
41        self.assertEqual(Address.objects.get().name,
                         'B. Botir')

42    def test_create_address_postalcode_too_long(self):
43        data = {"name": "C. Chai",
44                  "address": "Down Under 1",
45                  "postalcode": "ABCDEFGHIJ123456",
46                  "city": "Gotham",
47                  "country": "Northland",
48                  "email": chai@upside.com
49                  }
50      response = self.client.post(self.url, data,
                                      format='json')
51      response.render()
52      self.assertEqual(response.status_code,
                           status.HTTP_400_BAD_REQUEST)
53      self.assertEqual(response.content,
                           b'{"postalcode":
                           ["Ensure this field has no more
                           than 20 characters."]}')
```

Line 30 defines the maximum length test with a maximum postal code of `ABCDEFGHIJ12345` in *line 33*. We expect this test to succeed, as asserted in *lines 39-41*.

The test for the too-long postal code starts at *line 42* with a postal code of `ABCDEFGHIJ123456` in *line 45*, which exceeds the maximum length by one character. As asserted in *lines 52-53*, we expect this test to result in a bad request and an error message.

Let's run the tests again and check the outcome. Hey – the third test fails and delivers this output:

```
self.assertEqual(response.status_code, status.
HTTP_400_BAD_REQUEST)
AssertionError: 201 != 400
```

What happened? We expected status code `400`, but we received code `201`, which implies that the RESTful API accepted the postal code and added the address. Time to debug. The requirements state that the postal code may be up to `15` characters long. But what did we declare in the model (`models.py`)? We set the postal code like this:

```
postalcode = models.CharField(max_length=20, blank=True)
```

Aha – here is the culprit: we specified `max_length` to be `20` instead of `15`. Let's correct this in `models.py`:

```
postalcode = models.CharField(max_length=15, blank=True)
```

We also need to change the assertion in *line 53* of `tests.py` because the error message changes as well:

```
52 ...
53      self.assertEqual(response.content,
                         b'{"postalcode":
                         ["Ensure this field has no more
                         than 15 characters."]}')
```

Okay – when we rerun the unit tests, we'll see that all tests succeed. Great – we found and fixed a bug early with unit testing. Rerunning automated unit tests is quick and easy and helps us improve the quality of our software. Now, let's see how end-to-end testing works and if it reveals any error in the following section.

End-to-end testing microservices

End-to-end testing is trying out an application as a whole to ensure it meets the requirements of the product owner and stakeholders. The focus of end-to-end tests is on the flow of actions and data through the application – for example, a flow that spans all actions from the subscriber entering an address until the same subscriber checking the confirmation email.

Primarily, end-to-end tests are the domain of software testers and users, but we, developers, also need to determine that our software works as a whole before we hand it over to software testers.

Generally, we execute end-to-end tests later in the development cycle because we need all application parts to be finished. If your application depends on complex interfaces, you might want to do end-to-end tests earlier to catch errors. In that case, consider creating mocks that simulate application parts that still need to be developed.

As for errors found with end-to-end tests, these tests generally reveal interface errors between the parts because we have already unit tested the individual parts.

And for creating end-to-end tests, we can choose between manual and automated tests. Automated tests seem more efficient, but they often require quite some time to develop. So, if you only need a couple of end-to-end scenarios that you run twice or thrice, consider manual execution because this can save you time and money.

On the other hand, if you need a wide range of long scenarios that you'll run repeatedly, consider automating them.

In this section, we'll create and perform a manual end-to-end test. In the following subsection, we'll automate an end-to-end test so that you're familiar with both types and able to apply them as needed.

Besides our overall knowledge of the application and the input of the product owner, *user stories* are an excellent resource for designing end-to-end tests. Let's take the user stories of our application and, therein, look for flows on which to base end-to-end tests. These are the user stories drawn up by the product owner:

- *As a new subscriber, I want to register for a subscription to a computer magazine so that I'll receive the magazine monthly at my specified address.*
- *As the Subscription Management team, we want newly entered subscription addresses to be matched with existing base addresses so that we'll register correct and unique addresses only.*
- *As the Subscription Management team, we want to confirm a new subscription by email so that the subscriber knows the subscription has succeeded and can verify the delivery address.*

From these stories, we derive a happy path flow with steps and its implementers like this:

1. A subscriber enters an address (user + Django app).
2. The application shows the success page to the subscriber (Django app).
3. The producer offloads a task to match the address to existing base addresses (Django app).
4. The matching worker retrieves a list of base addresses from a RESTful API and matches the address (Celery worker + RESTful API).
5. The matching worker creates a new address via the RESTful API (Celery worker + RESTful API).
6. The matching worker offloads a task to send a confirmation email (Celery worker).
7. The email worker sends the email (Celery worker).
8. The subscriber receives and checks the email (user).

We then turn this flow into the following manual end-to-end test:

Test preparation:

1. Start Redis or RabbitMQ.
2. Start the RESTful API and the Celery workers for the matching and email microservices.

Test steps:

1. Open a browser and navigate to `http://127.0.0.1:8000/subscription/`.
2. Enter a correct address.
3. Click **Subscribe**.

Test assertions and checks:

1. Establish that the application shows the success page.
2. Check via the MongoDB web interface that the application added the new address to the `address` collection.
3. Check the Celery terminal for the email sent and if this contains the required information about the subscription and the registered address.

Test completion:

1. Delete the added address via the MongoDB web interface.

Okay – let's execute this end-to-end test. We'll hold on until you're ready. Did you find an error? Sure you did. The interfaces (the task offloads) look okay, but the confirmation email lacks the subscriber's registered address. We could have caught this error during the unit tests, but since these tests are automated, it's more likely that we would find the email error during end-to-end testing.

Let's fix this error, which, on second thought, also appears to be an interface error because the task offload omits `street`. Open the `tasks.py` file with the Celery workers and modify the `send_email_task` function like this:

```
34 ...
35 def send_email_task(name, street, email):
36     send_mail(
         "Your subscription",
         f"Dear {name},\n\nThanks for subscribing to our
           magazine!\n\nWe registered the subscription at
           this address:\n{street}.\n\nAnd you'll receive
           the latest edition of our magazine within three
           days.\n\nCM Publishers",
           magazine@cm-publishers.com,
           [email],
           fail_silently=False,
     )
```

This fixes the worker part of the interface and the email content. Next, we adjust the task offload in *line 32* to fix the producer part of the interface:

```
31 ...
32 send_email_task.delay(address["name"],
                         address["address"],
                         address["email"])
```

Finally, we'll also include streets in the unit tests for the mail microservice in the `/django-microservices/subscription_celery/subscription/`

`tests.py` file:

```
25 ...
26 class MailTest(unittest.TestCase):
27    def test_send_email(self):
28        send_email_task.delay("D. Dhozos",
                                "Down the road 91",
                                "dhozos@upside.com")
```

When we rerun the end-to-end test, we see that we have fixed the error. Lovely – let's automate this test with Selenium in the following section.

Automated testing with Selenium

Selenium is renowned for testing web applications and allows us to test web pages. Selenium automatically performs actions a user would do and can simultaneously execute assertions. Furthermore, Selenium works with Chrome, Firefox, and Safari. Therefore, besides installing Selenium, we must install the driver software for the browser or browsers we want to test.

In this section, we'll install Selenium and the Chrome driver and then build an automated version of the end-to-end test. We only cover what is necessary to create basic Selenium tests. For detailed information about Selenium, check the documentation at `https://selenium-python.readthedocs.io/`.

Okay – let's install the necessary software:

1. Install Selenium with this command:

   ```
   $ pip install selenium
   ```

2. Navigate in your browser to `https://selenium-python.readthedocs.io/`
3. `installation.html#drivers` and follow the instructions to install a driver for your browser and platform.
4. Ensure the installed driver is in your `PATH` environment.

We're all set to automate the end-to-end test with Selenium and assume we installed the Chrome driver. Selenium stands on its own and works apart from Django. Therefore, we create a Python test script in the `django-microservices` directory outside the Django context. We'll base this test script on the `unittest` package, and the script will have this functionality cover the steps from the end-to-end test:

- `unittest` functions to set up and tear down the test
- Selenium actions to automatically open a browser, navigate to the subscription page, and enter a subscription
- Selenium assertions to ensure the subscription was entered correctly into MongoDB
- `pymongo` code to delete the entered address so that the database remains clean

To accomplish this, create a file called `end_to_end_test.py` inside the `django-microservices` directory with this code:

```
 1 import unittest
 2 from pymongo.mongo_client import MongoClient
 3 from pymongo.server_api import ServerApi
 4 from selenium import webdriver
 5 from selenium.webdriver.common.by import By
 6 from selenium.webdriver.chrome.options import Options
 7 from time import sleep
 8
 9 class SubscriptionTest(unittest.TestCase):
10    def setUp(self):
11        options = Options()
12        options.headless = False
13        self.driver = webdriver.Chrome(options=options)
14        self.driver.get("http://127.0.0.1:8000/
                        subscription/")
15        self.assertIn('Subscription', self.driver.title)

16        con = "mongodb+srv://django-microservice:
                <password><cluster>/
                ?retryWrites=true&w=majority"
17        client = MongoClient(con,
                 server_api=ServerApi('1'))
18        db = client["Subscription"]
19        self.col = db["address_api_address"]
20
21    def test_create_subscription(self):
22        driver = self.driver
```

```
23          col = self.col
24
25          driver.find_element(By.NAME,
                "name").send_keys("S. Selenium")
26          driver.find_element(By.NAME,
                "address").send_keys("Earth avenue 301")
27          driver.find_element(By.NAME,
                "postalcode").send_keys("GALX 7")
28          driver.find_element(By.NAME,
                "city").send_keys("Rockdale")
29          driver.find_element(By.NAME,
                "country").send_keys("Everland")
30          driver.find_element(By.NAME,
                "email").send_keys("sel@earth.com)
31          driver.find_element(By.NAME,
                "subscribe_button").click()
32          sleep(2)
33          success_header = driver.find_element(By.NAME,
                "success_header").text
34          self.assertIn("Thanks!", success_header)

35          for last in col.find().sort('id', -1).limit(1):
36              self.assertEqual(last['name'], 'S. Selenium')
37              col.delete_one({'id': last['id']})

38      def tearDown(self):
39          self.driver.close()
40
41  if __name__ == "__main__":
42      unittest.main(verbosity=2)
```

As for the imports, we focus on the Selenium imports. *Line 4* imports the `webdriver` class, which steers the browser and user simulation actions. *Line 5* imports the `By` class, which helps us address web page fields and buttons, as we'll see in a moment. *Line 6* imports the `Options` class, which we'll use to open Chrome visually to see `webdriver` operate–otherwise, `webdriver` would operate headless without showing the browser window.

Then, we have the `SubscriptionTest` class on *line 9*. Within the `setUp` method, *lines 11-13* open a visual instance of Chrome with Selenium. *Line 14* navigates to our subscription app, and in *line 15*, Selenium checks if the subscription page is active by looking for the word `Subscription` in the page title. *Lines 16-19* set up a connection to the MongoDB database and collection.

Next, we have the `test_create_subscription` method on *line 21* . *Lines 22-23* enable the driver and the collection for the method. *Lines 25-30* fill in the address fields through Selenium commands such as `driver.find_element(By.NAME, "name").`

`send_keys("S. Selenium").`

The `driver` command chains the `find_element` and `send_keys` methods. First, `find_element` looks up an HTML element named `name`, and then `send_keys` fills this element with `S. Selenium`. The names of the HTML elements correspond to the names we set in `forms.py` for the address fields, `subscription.html` for the **Subscribe** button, and the `Thanks!` text in `success.html`.

Line 31 clicks the **Subscribe** button and submits the address. Then, the script waits 2 seconds in *line 32* for the microservices to finish. After this, *lines 33-34* ensure the success page is shown. *Lines 35-36* check that the address was added, and *line 37* deletes it.

Finally, *lines 38-39* tear down the test and close the driver, and *line 42* sets the `unittest` output to verbose.

Now, let's see Selenium in action. Run the test script as a standard Python script and watch how Selenium opens a Chrome instance, fills the subscription page, and closes Chrome. Also, check the test script output, which should show, "`1 test ran OK`". Finally, check in MongoDB the test script correctly deleted the added address.

This is cool because Selenium and Python do the work for us, and we can repeat this script at every application update to ensure we have everything intact.

Okay – so much for testing microservices. You can now unit- and end-to-end test microservices, including automated tests with `unittest` and Selenium. With this, you'll deliver software you can be proud of and confident in.

Summary

In this chapter, we learned about testing microservices applications to ensure our developed software meets the requirements set by the stakeholders. We walked through the principles of testing microservices and then started testing the individual software components by unit testing. When we covered the components, we mastered end-to-end testing to try out the application as a whole and determine if it worked as expected. Finally, we automated an end-to-end test with Selenium.

This gives you control over the quality of the software you deliver and enhances your reputation as a developer. Moreover, you save your organization costs by finding and fixing errors as soon as possible.

In the next chapter, you'll learn how to run the Django microservices and RESTful API as Docker containers, which makes our microservices application even more robust, independent, transportable, and scalable.

8

Deploying Microservices with Docker

To maximize microservices benefits, they should run in a stable, protected, platform-independent environment, and Docker containers are the ideal solution for this. **Docker containers** are isolated environments for software, providing a reliable runtime environment for applications, RESTful APIs, and microservices. So, Docker containers are perfect for Django microservice applications to be robust and scalable.

In this chapter, you'll learn what Docker is and how it works. Furthermore, you'll learn about containerizing microservices and the role of Docker Compose when deploying multiple containers. Finally, you'll explore the concepts and solutions for scaling microservices.

By the end of this chapter, you'll know how to build and deploy a microservices application in Docker.

To accomplish this, this chapter addresses the following topics:

- Introducing Docker
- Containerizing microservices
- Applying multi-container deployment with Docker Compose
- Deploying a Django microservices application
- Scaling microservices

With Docker containers, your Django microservices get the stable and scalable runtime environment they need to fulfill the requirements of modern web applications. So, jump in to deliver cutting-edge applications to your stakeholders.

Technical requirements

You can find the Docker file samples for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter08`.

Introducing Docker

Docker is an open-source tool for containerizing applications with these main parts:

- **Docker Images**: They encapsulate an application and its dependencies, like the operating system and Python packages. Images act as blueprints for the containers.
- **Docker Containers**: They encapsulate and expose an application based on an image.
- **Docker Engine**: It runs the containers.

Images are software package blueprints encapsulating an application and dependencies like Python packages. They are built or composed through the Docker Desktop interface or the **Docker Command Line Interface** (**Docker CLI**) and based on specifications from a `Dockerfile` or `docker-compose.yaml` file. In the subsection, *Containerizing microservices*, we'll look at the `Dockerfile`, and in the subsection, *Applying multi-container deployment with Docker Compose*, we'll explore the `docker-compose.yaml` file.

Containers are software packages that contain all parts to run in any environment. They can run anywhere because they *virtualize* the operating system. Docker runs containers from images, which are the software package blueprints.

Finally, the Docker Engine runs the containers so users can interact with them, and they perform processes like updating a database or creating a report.

If we apply the Docker parts to the sample subscription Django app, we get a set up like this:

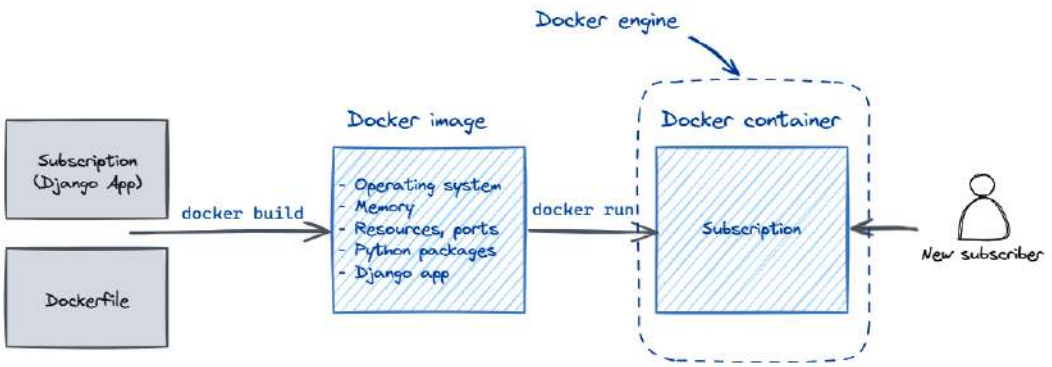

Figure 8.1 – Docker image and container

The subscription app runs in its own optimized environment without interference from, for instance, a database or other servers because these also run in their own isolated environment. But containers have other benefits, as we'll see in the following subsection.

Benefits of Docker (containers)

Docker is popular for software development and deployment and has become an industry standard due to these benefits:

- *Isolation*: Docker containers run isolated from other containers, ensuring reliability across different environments.
- *Efficiency*: Docker containers share resources, leading to efficient resource utilization.
- *Portability*: Docker containers run consistently across various environments, simplifying deployment and ensuring applications behave the same way in any environment.
- *Microservices architecture support*: Docker is well-suited for microservices architecture because each service runs in its own container, enabling optimal scaling and maintenance of microservices-based applications.
- *Scalability*: Docker allows application scaling by replicating containers across multiple hosts.
- *Security*: The isolation of Docker containers enhances security.

With these advantages, containerizing Django microservice applications makes sense. We'll do this step by step, from containerizing a microservice to multi-container deployment to deploying a microservices application. Let's start with containerizing in the following subsection.

Containerizing microservices

Containers are software packages that contain all parts to run in any environment. For example, a containerized version of the Django subscription app looks like this:

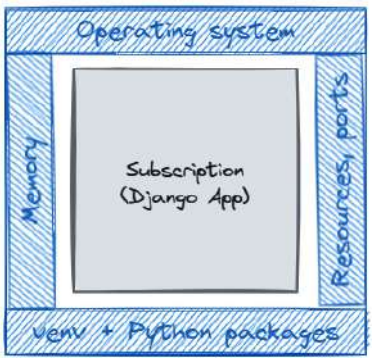

Figure 8.2 – Docker container and dependencies

The container virtualizes an operating system and enables memory, ports, and a virtual Python environment with the required packages for the Django app. Together, they form an isolated software package that runs on any host platform that supports Docker. So, we can quickly run the same container on Linux or Windows but also Azure or AWS.

To bring out the benefits of containers, it's best to have only one software package per container. You could have an app and, for instance, a PostgreSQL database running in the same container, but then you have the mutual risk of interference, which decreases the reliability. So, it's best to give the app and the database its own containers and connect them through a Docker network bridge:

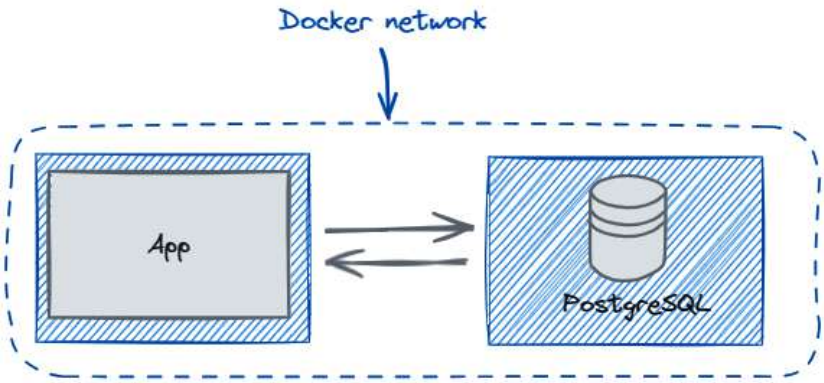

Figure 8.3 – Separated containers and network

Creating a container is a two-step process:

1. Build an image that contains the software and its dependencies.
2. Run a container from the image.

Let's focus on these steps and start building an image for our Celery-based subscription Django app. To build an image, Docker needs a `Dockerfile` that specifies what the app requires to run–for example, a particular Python version.

Open VS Code, and in the upper `subscription_celery` directory, create a file named `Dockerfile` with this content:

```
1 FROM python:3.11.6
2 ENV PYTHONDONTWRITEBYTECODE 1
3 ENV PYTHONUNBUFFERED 1
4 WORKDIR /
5 COPY requirements.txt ./
6 RUN python -m pip install --no-cache-dir -r
7             requirements.txt
8 COPY . .
```

```
9 EXPOSE 8000
10 ENTRYPOINT ["python3"]
11 CMD ["manage.py", "runserver", "0.0.0.0:8000"]
```

Line 1 sets the required Python version for the image.

Lines 2-3 specify environment settings. `PYTHONDONTWRITEBYTECODE` prevents Python from creating Python cache files, and `PYTHONUNBUFFERED` makes Python send its output directly to the terminal without buffering.

This settles Python, and now it's the packages turn. *Line 4* sets the working directory, *line 5* copies the `requirements.txt` file, and *line 6* installs the required packages.

Then, *line 8* virtually copies the Django app's directories and files into the image.

Finally, Docker starts the app. To do so, *line 9* sets the app's port to `8000`, and *lines 10-11* fire up the app.

If we built the image now, it would include all the Django app's directories and files, as well as a `venv` directory or a `.gitignore` file if we use Git. However, including the `venv` directory is undesirable because Docker will create the virtual environment itself, and a `.gitignore` file has no purpose inside an image and container. To keep our images as lean as possible, we use a `.dockerignore` file to exclude unnecessary directories and files.

To exclude directories and files from images, create a file named `.dockerignore` in the upper `subscription_celery` directory with these entries:

```
/__pycache__
/.venv
/.gitignore
/.vscode
/docker-compose*
/compose*
/Dockerfile*
LICENSE
README.md
```

Now, Docker ignores the specified directories and files. You can always add directories and files to this file and recreate an image if needed.

Important note

Docker images for message brokers like Redis and RabbitMQ and databases like PostgreSQL are so often used in containers that their suppliers created standard versions for them. So, we don't need to build these images and can run containers from them immediately, like we did in *Chapter 3*, *Setting Up the Development and Runtime Environment*. For example, we created a RabbitMQ container with this command:

```
$ docker run -it --rm --name my-rabbitmq
--network my-network -p 5672:5672 -p 15672:15672
rabbitmq:3.8-management-alpine
```

This container applies the standard `rabbitmq:3.8-management-alpine` image, and Docker downloads it automatically when creating the container.

Okay, we're ready to create and test the container with these steps:

1. Build the image with the following command from the upper `subscription_celery` directory:

   ```
   $ docker build . -t subscription-img
   ```

   The dot (`.`) includes all directories and files. And the `-t` (tag) option sets the image name to `subscription-img`. When you hit *Enter*, Docker starts installing the packages, as you see in the terminal output. And after a while, Docker finishes the image.

2. With the image created, we can now run a container from it with this command:

   ```
   $ docker run --name subscription-con –d -p 8000:8000
   subscription-img
   ```

   The `--name` option specifies the container name. The `-d` option detaches the container to run in the background, and the `-p` option sets the ports to `8000`. The command end with the image (`subscription-img`) that serves as the basis for the container.

3. To test the container, open your browser and navigate to `http://127.0.0.1:8000/subscription/`. Now, the subscription page opens, which proves the container runs correctly.

   This only runs the Django subscription app, not our entire microservices application. If you enter an address and click **Subscribe**, the container exits with an error because the Celery workers aren't running.

In the following subsection, we'll dockerize the whole microservices application into multiple containers and test it end-to-end.

Applying multi-container deployment with Docker Compose

In addition to building separate containers for each application component, we can also build multiple containers at once for the application components. This simplifies deployment and saves us time.

To containerize our microservices application, we distinguish the following components as potential containers:

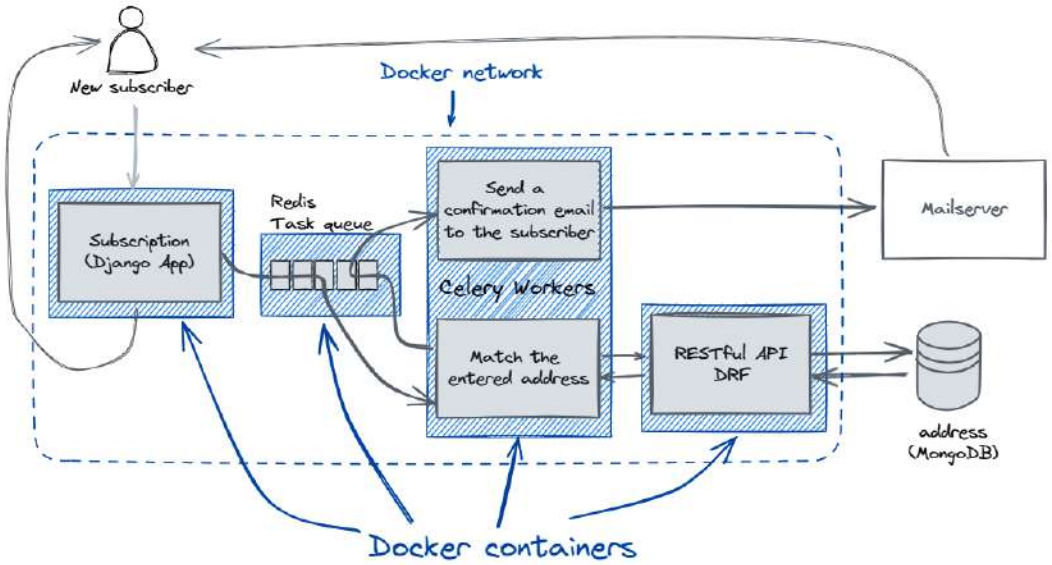

Figure 8.4 – Multi-container microservices application

This way, we'll have isolated containers for:

- The Django subscription app running at port `8000`.
- Redis as the message broker for Celery.
- The Celery workers for matching an address and sending a confirmation email.
- The RESTful API running at port `7000`.

We could also run MongoDB in a container, but since we use the cloud version, we have already isolated the database.

We could create these containers individually. But with the Docker compose facility, we can create the containers and necessary images in one go by what we call multi-container deployment. Still, we make an exception for the RESTful API because it will run in a separate, optimized environment.

Let's start by composing the multi-container deployment. To do so, Docker needs a `docker-compose.yaml` file in addition to the `Dockerfile`. The `docker-compose.yaml` file specifies the various containers.

To define the required containers, create a file named `docker-compose.yaml` in the upper `subscription_celery` directory with this content:

```
1 services:
2   web:
3     container_name: subscription_app
4     build: .
5     ports:
6       - 8000:8000
7
8   redis:
9     container_name: redis_celery
10    image: redis
11
12   worker:
13    container_name: celery_worker
14    build: .
15    entrypoint: celery
16    command: -A subscription_celery worker
                --loglevel=info
```

The file specifies each container as a service under *line 1*.

Lines 2-6 define the container for the Django subscription app with a `container_name` and `ports`. The `build` element addresses that all directories and files are included.

Then, *lines 8-10* specify the Redis container based on the standard `redis` image.

Finally, *lines 12-16* define the container for the Celery workers, where the `entrypoint` element refers to the `celery` command and the `command` element to the related options.

Before composing the multi-containers, we must ensure that Celery and Redis have the correct settings in the `settings.py` file. Because Redis and the Celery will run in separate containers, the `localhost` setting isn't applicable, and we must replace the Celery settings like this:

```
CELERY_BROKER_URL = "redis://redis:6379"
CELERY_RESULT_BACKEND = "redis://redis:6379"
```

Okay, now we can compose the containers and images in one go with the `compose up` command:

```
$ docker compose up -d
```

The `-d` option makes the containers run detached in the background. When you press *Enter*, Docker builds the images and creates and starts the containers.

Open your browser and navigate to `http://127.0.0.1:8000/subscription/` to check if Docker created the containers. You'll see that the subscription page opens, which proves that the Django subscription container runs correctly.

The Redis and Celery workers containers run, too, but we can't test them yet because we need to create the RESTful API container first. Before we create this container, we need to know which Docker network to add the container to since all containers must use the same network to collaborate.

Since we didn't specify a custom network when composing the containers, Docker generated a network automatically. We can look up this network with the following command:

```
$ docker network ls
```

Check the output. In addition to the `my-network` network that we created in *Chapter 3*, *Setting Up the Development and Runtime Environment*, you see a network called `subscription_celery_default`, which is the network for our microservices application.

Now, we create the RESTful API container as we did in the previous subsection, *Containerizing microservices*. We'll start with creating this `Dockerfile` in the upper `subscription_apis` directory:

```
FROM python:3.11.6
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
WORKDIR /
COPY requirements.txt ./
RUN python -m pip install --no-cache-dir -r
  requirements.txt
COPY . .
EXPOSE 7000
ENTRYPOINT ["python3"]
CMD ["manage.py", "runserver", "0.0.0.0:7000"]
```

Because the RESTful API runs at port `7000`, we `EXPOSE` this port and set it through the `CMD` option.

Next, copy the `.dockerignore` from the upper `subscription_celery` directory into the upper `subscription_apis` directory.

Finally, create the RESTful API container with these steps:

1. Build the image with the following command from the upper `subscription_apis` directory:

```
$ docker build . -t address-api-img
```

2. Create and run the container from it with this command:

```
$ docker run --name address-api --network subscription_celery_
default -d address-api-img
```

   The `--network` option ensures that the RESTful API container is part of the Docker network, making it accessible to the other microservices application parts.

3. Now, we can test our subscription application end-to-end. Let's first do this by hand. Navigate to the subscription page and enter a new address. Then, check the MongoDB web interface for the new address. Cool, isn't it? Our whole microservices application runs in isolated containers.
4. Finally, run the automated Selenium-based end-to-end test from *Chapter 7*, *Testing Microservices*, to make sure everything works as expected.

Okay, we successfully dockerized our microservices application, moving it from the development to the test stage. In the following subsection, we'll elaborate on deploying a Django microservices application with Docker.

Deploying a Django microservices application

We already covered the basic steps for deploying a Django microservices application with Docker in the previous subsections, *Containerizing microservices*, and *Applying multi-container deployment with Docker Compose*. However, the deployment also concerns updating images and containers in case of new microservice releases.

We can deploy microservice releases through either Docker Desktop or Docker CLI. Because the Docker CLI commands are the same for all Docker-supported platforms, we'll explore the most common Docker commands for deploying our microservice applications.

Docker CLI provides an extensive list of commands (`https://docs.docker.com/engine/reference/commandline/docker/`). For containers alone, Docker has 25 commands. There are far too many to cover here, so we will look at the commands that cover these common deployment steps:

- Show a list of the images we created.
- Show a list of created and running containers.
- Inspect the console output of a container to ensure it works correctly.
- Stop a container.
- Start a container.

- Remove a container.
- Remove an image.

We'll first examine the commands for these steps and then look at a basic workflow for deploying a new microservices version.

> **Important note**
> For these commands, we assume that the containers from subsection, *Applying multi-container deployment with Docker Compose*, are still running.

Let's start by determining our baseline and seeing what images and containers we have running.

Showing a list of the images we created

Over time, we can have dozens of images and quickly lose track. But with this command, we always have an overview of our images:

```
$ docker image list
```

Or these shortcut versions:

```
$ docker image ls
$ docker images
```

Either command shows details like creation time and size. For further options, check `https://docs.docker.com/engine/reference/commandline/image_ls/`.

Next, let's look up our containers.

Showing a list of created and running containers

Just like images, we can quickly lose track, and with this command, we show a list of running containers:

```
$ docker container list
```

Or its shortcut version:

```
$ docker ps
```

The list shows details like status, ports, and the underlying image. However, this list only shows running containers and not the containers we stopped or that crashed. Use `-a` option to show all containers:

```
$ docker ps -a
```

And you'll get a list of all your Docker containers. The status shows when we stopped a container.

For further options, check `https://docs.docker.com/engine/reference/commandline/container_ls/`.

Now, let's ensure that a container works correctly.

Inspecting the console output of a container

When we run a microservice from a terminal, we can directly check its output. However, we don't have a terminal to inspect if we run a microservice as a Docker container. Nevertheless, we can check the console output of a container with this command:

```
$ docker logs <container_name>
```

You may know the container's name by heart; otherwise, you can look it up with the `docker ps` command. When executing the `docker logs` command, you'll see the output as you would from a terminal, so you can ensure the microservice works correctly.

For example, to check the Django subscription app, enter:

```
$ docker logs subscription_app
```

For further options, check `https://docs.docker.com/engine/reference/commandline/container_logs/`.

Next, let's stop a container.

Stopping a container

If you want to remove a running container, you need to stop it first with this command:

```
$ docker stop <container_name>
```

After stopping, you can check the container's status with the `docker ps -a` command. Furthermore, if you enter the `docker ps` command, you won't see the container because you only list the running containers.

Now, it's your turn. Stop the `celery-worker` container. The command output returns the container name, indicating the container stopped.

For further options, check `https://docs.docker.com/engine/reference/commandline/container_stop/`.

Our microservice application is now hampered because we stopped one of its containerized parts, so let's restart the container in question.

Starting a container

Images are container blueprints that are always available, but containers need to be started after creation. When creating containers, Docker automatically starts them, but over time, containers stop running due to, for example, system reboots. In such cases, we can start a container with the following command:

```
$ docker start <container_name>
```

After starting, you can enter the `docker ps` command to ensure the container runs.

Now, you bring up the `celery-worker` container again. The command output returns the container name, indicating the container started.

For further options, check `https://docs.docker.com/engine/reference/commandline/container_start/`.

Next, let's remove a container.

Removing a container

If we update a microservice, we must also update its corresponding Docker container because its image changed. This takes us to remove the old container and create a new version. Enter this command to remove a container:

```
$ docker rm <container_name>
```

After removing, you can enter the `docker ps -a` command to ensure the container has disappeared.

If all goes well, you'll still have the `subscription-con` container running from the subsection, *Containerizing microservices*. Stop this microservice first, then remove it. The command output returns the container name, indicating Docker removed the container.

For further options, check `https://docs.docker.com/engine/reference/commandline/container_rm/`.

Next, we'll address our last common deployment step: removing an image.

Removing an image

When updating a microservice, we must also update its corresponding Docker image because the microservice blueprint contains the software and dependencies. This requires removing the old image and build a new version. Enter this command to remove an image:

```
$ docker rmi <image_name>
```

After removing, you can enter the `docker image ls` command to ensure the image has disappeared.

Now, you remove the `subscription-img` image. The command output returns that Docker removed the latest version of the image.

For further options, check `https://docs.docker.com/engine/reference/commandline/image_rm/`.

Okay, so far, for the Docker commands. Now, let's examine a workflow for deploying a new version of our microservices application using these commands.

Deploying a new microservices version

When we update our microservices application, we must recreate its docker images and containers. With the Docker commands for removing images and containers, we can quickly execute a **deployment workflow** with these command steps:

Action	Command step
Stopping containers	1. `$ docker stop subscription_app` 2. `$ docker stop celery_worker` 3. `$ docker stop redis_celery`
Removing containers	1. `$ docker rm subscription_app` 2. `$ docker rm celery_worker` 3. `$ docker rm redis_celery`
Removing images	1. `$ docker rmi subscription_celery-web` 2. `$ docker rmi subscription_celery-worker`
Recreate images and containers	1. `$ docker compose up -d`

Table 8.1 – Docker deployment workflow

Commands 1-6 stop and remove the containers.

Commands 9-10 remove the images. We don't remove the Redis image because it's the standard version we applied and downloaded earlier, and we'll use this image again.

Finally, *command 11* recreates the images and containers, so we have deployed our updated microservices application.

Overall, this requires quite some typing, but we can simplify it by creating a Shell script that includes these workflow commands. You can find an example of such a script at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/blob/main/Chapter08/subscription_celery/deployment.sh`.

Important note

A simple and short alternative for the deployment workflow is executing the `docker compose up` command with the `--force-recreate` option. This option forces Docker to recreate the images and containers. However, for some unknown reasons, the `--force-recreate` option doesn't always work correctly, and therefore, the workflow is a reliable deployment method.

Okay, we now know how to dockerize our microservices application, including updating a new version. But Docker has more features.

If you want more information, check books like *Docker for Developers* by Richard Bullington-McGuire, Andrew K. Dennis, and Michael Schwartz and *Hands-On Docker for Microservices with Python* by Jaime Buelta. And you can also find more information on Docker at `https://docs.docker.com/`.

From a Docker container perspective, there's one more topic to cover, and that's scaling microservices, which we'll address in the following subsection.

Scaling microservices

In general, scaling is the ability of a system to process more workload while maintaining performance. For microservices, this means that when the number of users expands, scaling ensures the microservices still perform as expected.

Technically, **scaling** means (dynamically) adding the capacity of resources like available memory, processors, and network bandwidth for microservices. Therefore, scaling is primarily the domain of the system administrator. However, developers should have conceptual knowledge of scaling methods and know about scaling solutions to work with system administrators and ensure the smooth running of microservices.

Systems administrators can either scale horizontally or vertically. In this subsection, we'll explore these methods and look at solutions to scale containerized microservices. We'll start with horizontal and vertical scaling. Next, we'll look at Docker Swarm and Kubernetes as scaling solutions.

Vertical and horizontal scaling

Scaling is increasing the available resources for microservices, and it can be done horizontally or vertically. Let's start with vertical scaling because it is the quickest method and is usually performed first.

Vertical scaling means adding memory, processors, or disk space to the server where our microservices run:

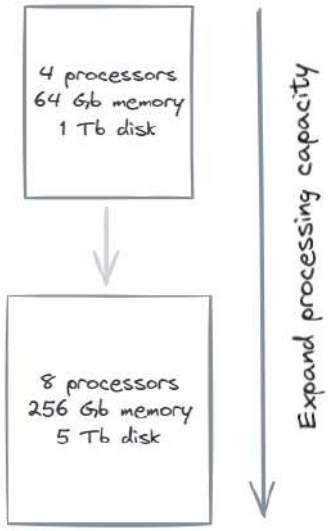

Figure 8.5 – Vertical scaling by adding resources

If the limit of vertical scaling is reached or we know in advance that our application will have many users, horizontal scaling can increase the processing throughput. **Horizontal scaling** means expanding the number of servers where our microservices run:

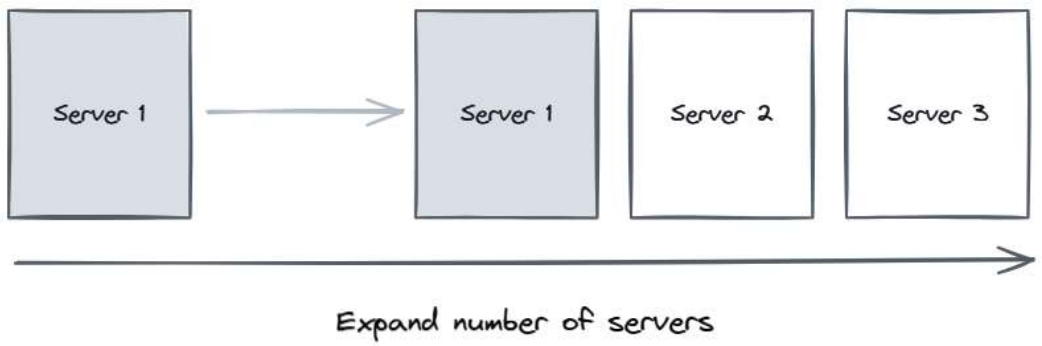

Figure 8.6 – Horizontal scaling by adding servers

In this context, a server doesn't need to be a physical server but can also be a virtual server in the cloud in Azure or on ASW. It can also be a node or a pod if we look at horizontal container scaling solutions like Docker Swarm or Kubernetes, as we will discuss in the following subsections.

Docker Swarm

Docker Swarm is a container orchestration system that allows horizontal scaling. Within Docker Swarm, there are multiple Docker engines that will enable multiple containers for our microservices. For example, instead of one container and instance of our address-matching microservice, we can have various address-matching microservices working simultaneously:

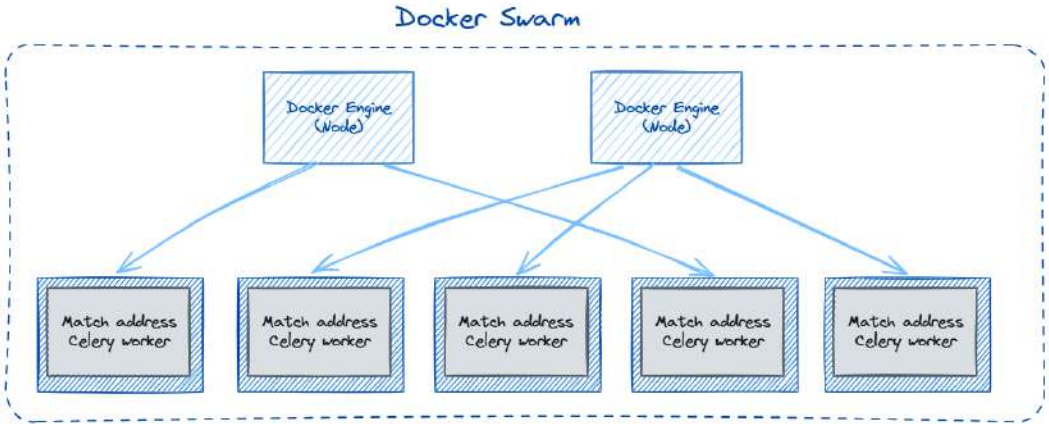

Figure 8.7 – Docker Swarm architecture

The Docker Engines are called nodes within Docker Swarm, and they distribute tasks among workers, allowing many users to be served simultaneously while maintaining performance.

Kubernetes

Kubernetes is also a container orchestration system that allows horizontal scaling and is the standard for many microservices applications because it's highly extensible. Kubernetes also works with nodes, but it runs the workers in pods:

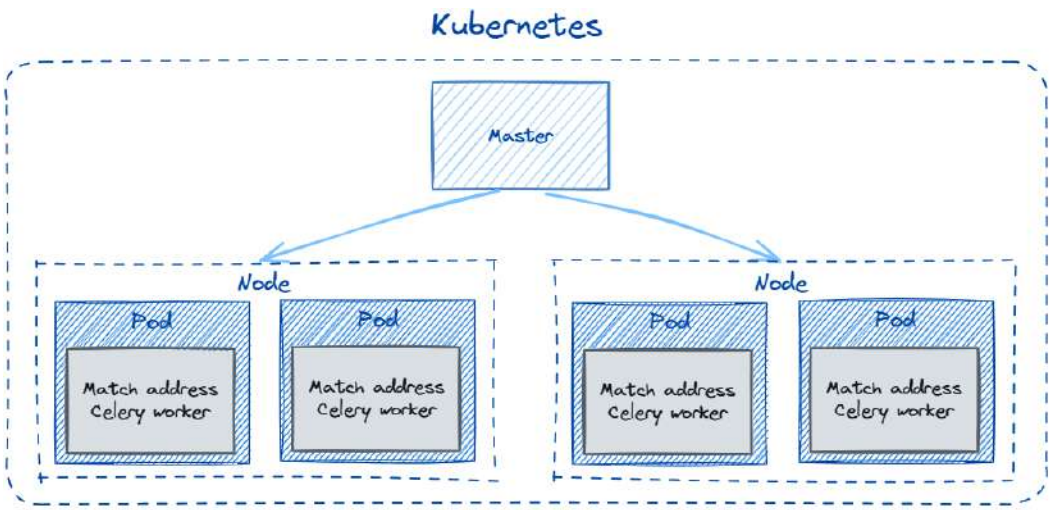

Figure 8.8 – Kubernetes architecture

The Kubernetes master and nodes orchestrate the task distribution and execution to serve high user loads.

Okay, this should be enough for us to set up scaling with system administrators. And with this, we finish our exploration of containerizing microservices.

Summary

In this chapter, we learned about containerizing microservices with Docker. First, we looked at what containerizing entails and the benefits it provides. Then, we learned how to containerize a single microservice and multiple microservices simultaneously through Docker Compose. Next, we set up a deployment workflow for containerizing our microservices application. We finished by exploring the concepts and solutions for scaling containerized microservices.

With this, you can deliver stable and scalable Django microservices that fulfill the requirements of modern web applications.

This concludes *Part 2*, *Building the Microservices Foundation*, in which we mastered developing Django microservices and laid the foundation for building professional microservices applications. From there, we move on to *Part 3*, *Taking Microservices to the Production Level*, where we'll address topics like security and caching, further enhancing our microservices to the production level. And where we'll also look at best practices and how to transform a monolithic Django web application into a microservices version.

In the next chapter, we'll add security measures to our microservices to withstand the ever-increasing threats of cyber-attacks.

Part 3: Taking Microservices to the Production Level

In this part, you'll learn how to make microservices suitable for production. First, you'll learn how to protect microservices against cyber-attacks. Then, you'll master creating microservices to maintain or improve their performance. After that, you'll learn some best practices to benefit from earlier practice. Finally, you'll learn how to transform an existing monolithic application into a microservices version.

This part contains the following chapters:

- *Chapter 9*, *Securing Microservices*
- *Chapter 10*, *Improving Microservices Performance with Caching*
- *Chapter 11*, *Best Practices for Microservices*
- *Chapter 12*, *Transforming a Monolithic Web Application into a Microservices Version*

9

Securing Microservices

Because of the isolated nature of Django **microservices** and the controlled containerized environment in which they run, microservices are already implicitly quite secure against intrusion. However, depending on the sensitivity of the processed data, you might want to secure your microservices further.

Depending on the security requirements, securing microservices can range from **token authorization** to **external authentication** via an identity provider. We focus on token authorization because this gives you insight into securing microservices and a good foundation if you need more advanced solutions, such as external authentication.

In this chapter, you'll learn about the security options for microservices and how to apply token authorization for your microservices.

By the end of this chapter, you'll know about the possible security measures between microservices and calling clients or producers such as a Django app. You'll also know about security measures for when microservices call each other, and you'll be able to apply token authorization security for producers and microservices.

This chapter covers the following topics:

- Introducing microservices security
- Controlling client access to microservices
- Securing access between microservices

As developers, we're mainly focused on fulfilling the functional needs of our stakeholders. But in a world where cyber-attacks are commonplace, we must consider the security of our microservices and take action when necessary. So, follow along because then you will know how to secure microservices when required.

Technical requirements

We'll create and process authorization tokens in this chapter, and for that, we need the `PyJWT` package, which you install like this:

```
$ pip install pyjwt
```

We could include (hardcode) the created tokens in our Django app and microservices, but that would be rather unsafe. Therefore, we'll include the tokens as settings in a safe `.env` file, and to process the tokens, we'll apply the `python-dotenv` package, which you install as follows:

```
$ pip install python-dotenv
```

Furthermore, you'll find the files for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter09`.

Introducing microservices security

For securing microservices, we must consider these two situations and approaches:

- **Controlling external access**: The access to microservices from (external) clients or producers such as a Django app
- **Controlling inter-service access**: The access to microservices from other microservices such as the match address microservice calling the mail microservice

Because of its vertical flow direction, external access security is called **north-south security**. So, we use the term **east-west security** for horizontal inter-service access security. In the following subsections, we'll look at how these apply to the microservices architecture and start with north-south security.

North-south security for microservices

If application requirements state that producers (clients) must authenticate themselves when offloading tasks to workers (microservices), we can apply set-up north-south security. We call this type of security measurement north-south security because of the vertical direction, like in this diagram for our sample microservices application:

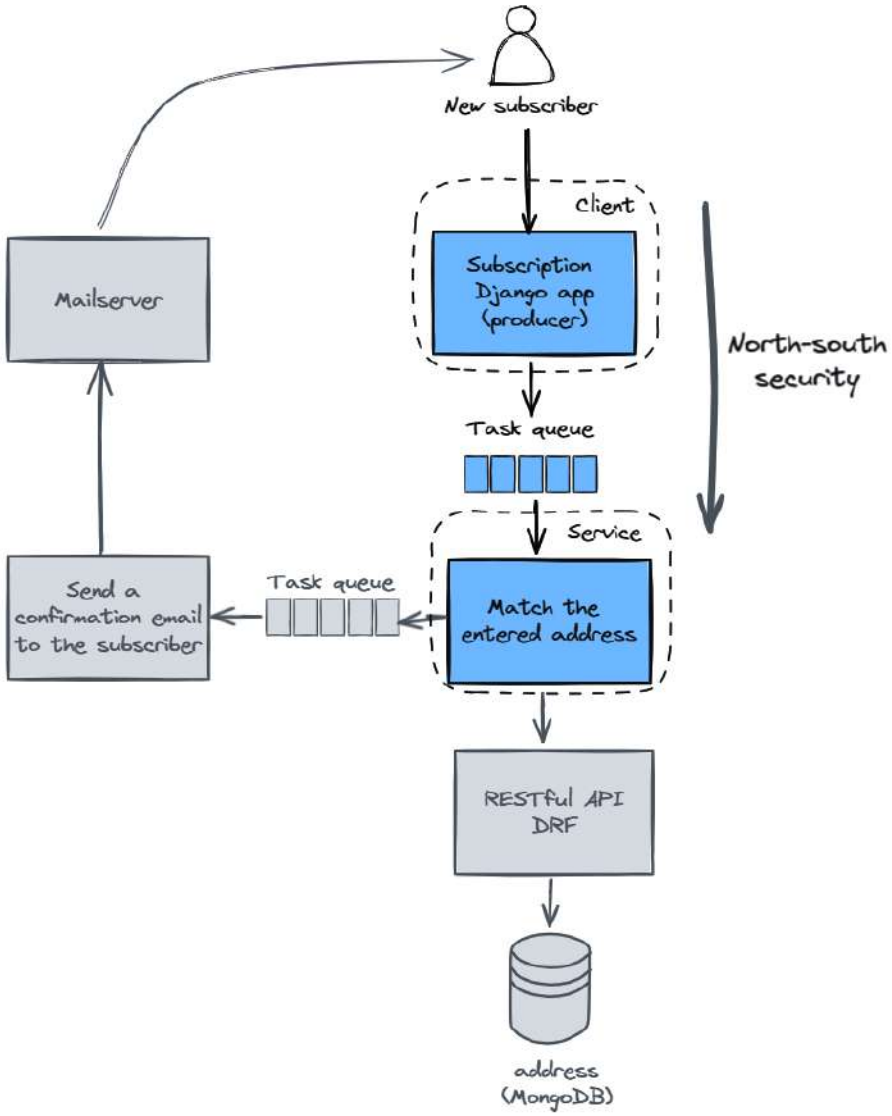

Figure 9.1 – North-south security from client to microservice

Here, the Django subscription app acts as the client, and the match address worker as the service. Instead of the open communication we have now, we require the Django app to authenticate itself to the match address worker, and the worker only executes the task if the authentication succeeds.

This ensures that only authorized clients can offload tasks to our microservices, but if application security requirements are very strict, we can go a step further, as we'll see next.

East-west security for microservices

If we run our microservices in the cloud, such as on Azure or AWS, or when our local microservices could be penetrated externally through the network, we might want to protect them from being called by other unauthorized microservices. To do so, we must apply east-west security, which is called that because of its horizontal direction, like in the following diagram for our sample application:

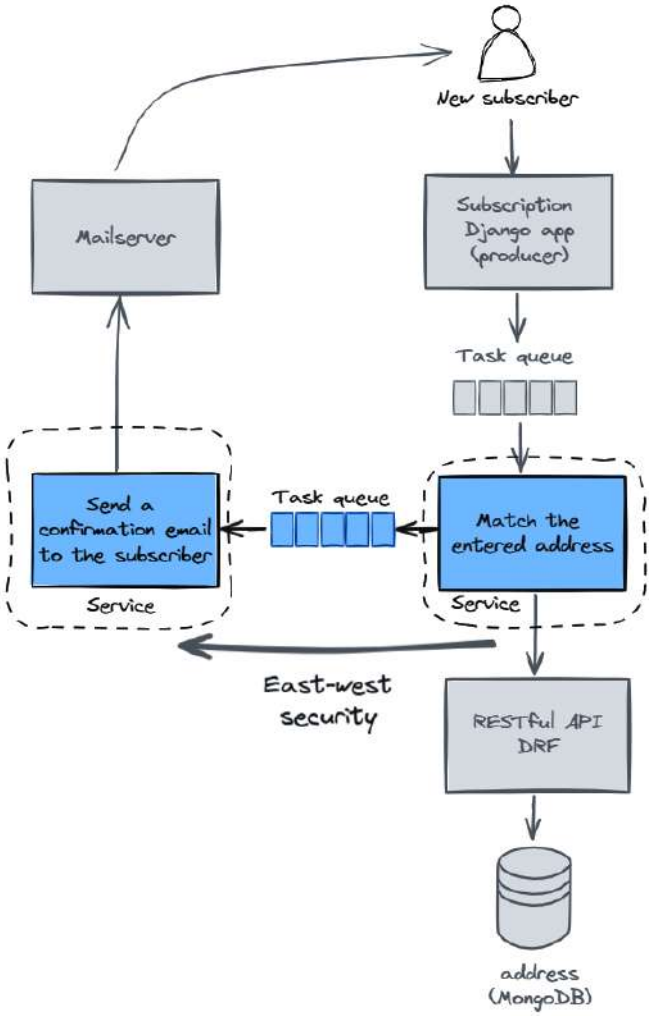

Figure 9.2 – East-west (inter-service) security between microservices

East-west security occurs between services, which is why we also call it **inter-service security**. Suppose we apply this for our sample application. In that case, the match address worker must authenticate itself to the send email worker, and the send email worker only sends the mail when authentication succeeds.

Once we have decided on the approaches, the next step is to select the technique to implement them. Basically, we have these types of security techniques:

- Token-based security such as **JSON Web Token** (**JWT**)
- User-based security such as **Open Authorization 2.0** (**OAuth 2.0**)

Let's explore these techniques further and start with JWT.

Token-based security with JWT

Tokens are pieces of digital information that represent the authenticity of a system or user. The most common token type is the **bearer token**, which is a piece of information, like a string, used to grant access. JWT is an example of a bearer token commonly used for authentication in web applications and, therefore, well suited for our sample application.

The JWT standard is based on JSON, and a JWT is a structured JSON object. It is a self-contained way to securely exchange data between a client and a service or two services. A JWT consists of these parts:

- **Header**: This contains the token type (JWT) and the signing algorithm
- **Payload**: This contains the statements about the calling client or service, such as a user or service name
- **Signature**: This contains `header`, `payload`, and a `secret` key.

Therefore, a generated JWT has this three-part format:

```
xxxx.yyyy.zzzz
```

The preceding format can be seen in this example of a JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJjbGllbnRfcHJvZHVjZXIiOiJzdWJzY3JpcHRpb25fYXBwIn0.
SMCUj11R4GahenNsKpJT340RddPS1U7JndsB4W_RCrs
```

For our sample microservices application, the JWT authentication process covers these steps:

1. We generate a client JWT with the `PyJWT` package by specifying `header`, `payload`, and the `secret` key.
2. In the Django subscription app, we add the generated JWT to the task message in the producer.
3. The receiving match address microservice worker parses the JWT, and the worker executes its task if the JWT validates.

We'll focus on token-based security for our sample application because this gives us insight into securing microservices and a good foundation when we need advanced solutions, such as OAuth 2.0, as we'll look into next.

User-based security with OAuth 2.0

OAuth 2.0 is a protocol that grants access on behalf of a user without sharing the user's credentials to the called service. To avoid sharing user credentials, the OAuth 2.0 scenario utilizes external authentication via an identity provider. The client sends its credentials to the identity provider to prove its identity.

When the identity provider accepts a client's credentials, it issues a token the client uses to authenticate itself to a service. As an example for our sample application, this process could cover these steps:

1. The Django subscription app calls an external identity provider and submits its credentials.
2. The identity provider checks the credentials and issues a token when the credentials are correct.
3. The Django subscription app receives the token and sends this with the task message to the match address worker.
4. The match address worker validates the token via the identity provider, and if the token is correct, the worker executes the requested task.

This is the most advanced security technique because it utilizes an independent third party (the identity provider) and requires internet access to the provider from the client and the microservice.

For our sample microservices application, we'll apply token-based security. In the following subsection, we'll implement north-south security for the collaboration between the Django subscription app and the match address worker. Then, in the subsequent subsection, we'll add east-west security for the collaboration between the match address and send emails worker.

Controlling access to microservices

We can apply north-south security to secure communication between producers (client) and workers (services) and to prevent unauthorized producers from executing our microservices:

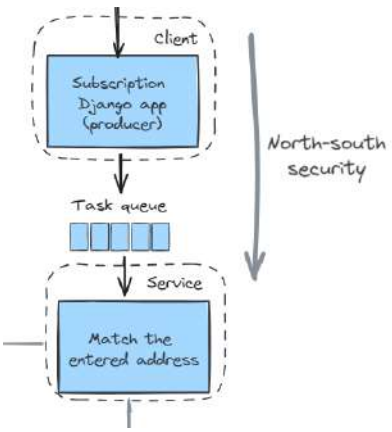

Figure 9.3 – Controlling north-south access to microservices

To see how this works, we'll set this up for the Django subscription app and the match address worker in our sample application with JWT.

We'll start by creating a JWT for the Django app. This is a one-time action for which we use a standalone Python script. We could hardcode the generated JWT into the Django app and the match address worker, but that's unsafe, so we'll include the tokens as settings in a safe `.env` file, which we'll process with the `python-dotenv` package. Follow the next steps to generate and store the JWT:

1. Create a file called `generate_token.py` in the `django-microservices` directory with this code:

```
1 import jwt
2
3 def create_jwt(payload_to_encode):
4     jwt_token = jwt.encode(
5         payload=payload_to_encode,
          key="Logic takes you from a
          to b, imagination
          takes you everywhere"
          algorithm="HS256")
6    return jwt_token
7
8 if __name__ == "__main__":
9    print(create_jwt({"client_producer":
                       "subscription_app"}))
10    print(create_jwt({"service_producer":
                       "match_address_worker"}))
```

   Line 1 imports the `PyJWT` package.

   Line 3 defines the `create_jwt` function, which receives `payload_to_encode` and returns `jwt_token` in *line 6* as created by *line 4*. Within *line 4*, the `key` parameter holds `secret` for `signature`, and the `algorithm` parameter holds the signing algorithm for `header`, in this case, `HS256`. `HS256` is a hashing algorithm that uses one secret.

   Finally, *line 9* calls the `create_jwt` function, provides the payload for the JWT, and prints the generated JWT. In our case, we set the payload's key to `client_producer` and its value to `subscription_app` since we want a JWT for the Django subscription app. However, you can compose any payload you like to fulfill your identification needs. Since this is a one-time script with no risk of unmasking, we hardcode the JWT key and algorithm.

   Likewise, *line 10* generates a JWT for the combination of `service_producer` and `match_address_worker`.

2. Run the `generate_token.py` script.

   * *The terminal output shows the generated JWTs.*

3. Now, create a file called `.env` inside the `subscription_celery/`
4. `subscription` directory with this content:

```
JWT_KEY= Logic takes you from a to b, imagination
    takes you everywhere
JWT_ALGORITHMS=HS256
CLIENT_PRODUCER=subscription_app
CLIENT_TOKEN=<JWT_TOKEN>
SERVICE_PRODUCER=match_address_worker

SERVICE_TOKEN=<JWT_TOKEN>
```

This `.env` file will be used by both the Django subscription app and the match address worker. Therefore, it will include the generated JWTs (`CLIENT_TOKEN` and `SERVICE_TOKEN`), the `secret` key (`JWT_KEY`), the algorithm (`JWT_ALGORITHM`), and the payload values (`CLIENT_PRODUCER` and `SERVICE_PRODUCER`) since we want to protect all these values within our sample application.

Replace `<JWT_TOKEN>` with your generated tokens.

Next, we'll update the Django app to send the JWT as part of the task message to the match address worker:

5. Open the `views.py` file in the `subscription_celery/subscription` directory and modify its content to this:

```
1 import os
2 from django.views.generic.base import TemplateView
3 from django.views.generic.edit import FormView
4 from dotenv import load_dotenv
5 from subscription.forms import SubscriptionForm
6 from subscription.tasks import match_address_task
7
8 load_dotenv()
9
10 client_token = os.getenv("CLIENT_TOKEN")
11
12 class SubscriptionFormView(FormView):
13     template_name = "subscription/
          subscription.html"
14     form_class = SubscriptionForm
15     success_url = "/success/"
16
17     def form_valid(self, form):
18         task_message = {
19             "name": form.cleaned_data["name"],
```

```
20            "address": form.cleaned_data["address"],
21            "postalcode": form.
22                 cleaned_data["postalcode"],
23            "city": form.cleaned_data["city"],
24            "country": form.cleaned_data["country"],
25            "email": form.cleaned_data["email"],
26            "client_token": client_token
27        }
28    match_address_task.delay(task_message)
29
30    return super().form_valid(form)
31
32 class SuccessView(TemplateView):
33    template_name = "subscription/success.html"
```

The Django subscription app only needs to send the JWT token as part of the task message. For this, the app reads the JWT token from the `.env` file. First, *line 1* imports the `os` package because the app will read the JWT as an environment setting via the `os.getenv` method in *line 10*. To enable the values from the `.env` file as secure environment settings, *line 4* imports the `python-dotenv` package, and *line 8* actually loads the values from the `.env` file as environment settings.

Finally, *line 26* adds the JWT token to the task message.

Now, we'll modify the match address worker so it checks whether a calling producer provided a valid token before execution.

6. Open the `tasks.py` file in the `subscription_celery/subscription` directory and change its introduction content to this:

```
1 import jwt
2 import os
3 import requests
4 from celery import shared_task
5 from django.core.mail import send_mail
6 from dotenv import load_dotenv
7 from rapidfuzz import fuzz
8 from subscription.models import Address
9
10 load_dotenv()
11
12 jwt_key = os.getenv("JWT_KEY")
13 jwt_algorithms = os.getenv("JWT_ALGORITHMS")
14 expected_client_producer =
       os.getenv("CLIENT_PRODUCER")
15 expected_service_producer =
```

```
        os.getenv("SERVICE_PRODUCER")
16 service_token = os.getenv("SERVICE_TOKEN")
17
18 def decode_token(token: str, caller_type: str) ->
   dict:
19    try:
20        decoded_token = jwt.decode(jwt=token,
                                     key=jwt_key,
                                     algorithms=
                                     jwt_algorithms)
21    except jwt.exceptions.InvalidSignatureError:
22        match caller_type:
23            case 'client':
24                return {"client_producer": "The
                      producer sent an invalid
                          token"}
25            case 'service':
26                return {"service_producer": "The
                      producer sent an invalid
                          token"}
27    return decoded_token
```

Lines 12-16 read the settings from the `.env` file. The `expected_client_producer` variable holds the client producer that the match address worker expects after decoding the JWT. If the client producer is as expected, the match address worker executes its task.

The decoding takes place in the `decode_jwt` function, which starts in *line 18*. This function receives `token` (JWT) and `caller_type`. For now, `caller_type` is only relevant to the client producer (Django app), but in the following subsection, we'll implement east-west security, and then we'll have a service producer as well. That is why we have provided the caller type already.

Regarding processing, the `decode_jwt` function uses a try-exception construction to decode (*line 20*) and return (*line 27*) the payload if the JWT is okay. If the JWT is invalid, the `except` block in *lines 21-26* returns an alternate payload, depending on the caller type. For now, only the client type is relevant, but we already foresee the service type because we need it in the next subsection to implement east-west security.

Next, we'll modify the `match_address_task` function to decode and check the calling producer.

7. Replace the `match_address_task` function with this version:

```
28 ...
29 @shared_task
```

```
30 def match_address_task(address):
31     calling_producer = decode_token(
           address['client_token'],
           'client')['client_producer']
32     if calling_producer ==
           expected_client_producer:
33         response = requests.get('http://address-
               api:7000/api/v1/addresses/')
34         addresses = [a_address['address'] for
               a_address in response.json()]
35         top_score = 0
36         min_score = 70
37         match_address = address["address"]
38         for base_address in addresses:
39             score = round(fuzz.ratio(
                           address["address"].lower(),
                           str(base_address).lower()))
40             if score >= top_score and score >=
                   min_score:
41                 top_score = score
42                 match_address = base_address
43             if top_score == 100:
44                 continue
45
46         print(f'Match address: {match_address} >
               Score: {top_score}')
17
48         address = {"name": address["name"],
               "address": match_address,
               "postalcode":
                   address["postalcode"],
               "city": address["city"],
               "country": address["country"],
               "email": address["email"]
               }
49         print(address)
50         response = requests.post('http://address-
               api:7000/api/v1/addresses/',
                   data=address)
51
52         print(f"New address inserted for
               {address['name']}")
```

```
53
54          send_email_task.delay(address["name"],
                address["address"],
                       address["email"])
55      else:
56          print(f"Authentication failed
                (match_address_task):
                    {calling_producer}")
```

First, the match address worker decodes the received JWT via the `decode_token` function in *line 31* and sets the decoded payload value for the `client_producer` key to the `calling_producer` variable to identify who issued the task message.

Then, *line 32* checks whether the `calling_producer` matches the `expected_client_producer`. If so, the address matching continues in *line 33*, as we implemented earlier. Only now do we send the request to the `address-api` URL instead of `localhost` because the RESTful API is now running in a Docker container. If the producers don't match, for example, because the JWT is invalid and the exception in *lines 21-26* occurred, the address matching skips, and *lines 55-56* print an error message.

Okay, this settles secure access to the match address worker from the Django subscription app. Let's redeploy the dockerized version of our sample application and test it.

8. Run the `./deployment.sh` script that we created in *Chapter 8*, *Deploying Microservices with Docker*.
9. Open your browser, navigate to the subscription app, and enter an address.
10. Check MongoDB for the added address.

Everything works as expected, but now with north-south security. This ensures that only authorized clients can call the match address worker. Next, let's add east-west security to secure inter-service communication as well.

Securing data communication between microservices

We can also use JWT to implement east-west security for safe collaboration between the match address and send email workers:

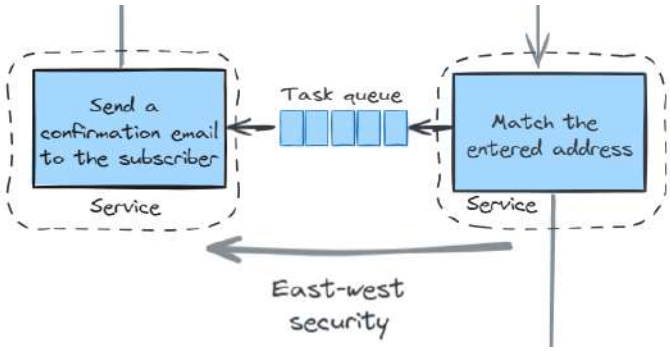

Figure 9.4 – Securing east-west access between microservices

This way, we ensure that an authorized co-worker can only call the send email worker. We build on the modifications we made in the previous subsection for controlling access to microservices, so if you still need to make these modifications, you should process them first.

Basically, the JTW-based east-west security is similar to the north-south implementation and takes these steps:

1. In the `tasks.py` file, replace the `send_email_task` function with this version:

```
57 ...
58 @shared_task
59 def send_email_task(name, street, email, token):
60     calling_producer = decode_token(token,
           'service')['service_producer']
61     if calling_producer ==
           expected_service_producer:
62         send_mail(
               "Your subscription",
               f"Dear {name},\n\nThanks for subscribing
               to our magazine!\n\nWe registered
               the subscription at this address:
               \n{street}.\n\nAnd you'll receive
               the latest edition of our magazine
               within three days.\n\nCM Publishers",
               magazine@cm-publishers.com,
               [email],
               fail_silently=False,
         )
63     else:
64         print(f"Authentication failed
               (send_email_task): {calling_producer}")
```

The send email worker decodes the received JWT (the `token` parameter in *line 59*) via the `decode_token` function in *line 60* and sets the decoded payload value for the `service_producer` key to the `calling_producer` variable to identify which service issued the task message.

Line 61 checks whether `calling_producer` and `expected_service_producer` match. If so, *line 62* sends the email. If the producer match fails, the email sending skips, and *lines 63-64* print an error message.

Next, we'll need to modify the task call from the match address worker to the send email worker to accommodate the JWT token.

2. Replace *line 54* in `tasks.py` with this line to add the `service_token` parameter:

```
53 ...
54         send_email_task.delay(address["name"],
               address["address"],
                   address["email"],
                       service_token)
55 ...
```

This completes the east-west security for our microservices. As the final step, we'll redeploy the dockerized version of our sample application and test whether the security measurements work.

3. Run the `./deployment.sh` script we created in *Chapter 8*, *Deploying Microservices with Docker*.
4. Open your browser, navigate to the subscription app, and enter an address.
5. Check MongoDB for the added address.

Our sample microservices application is now secured by north-south and east-west security. This ensures that only authorized clients can call the match address worker, and only authorized services can call the send email worker. With this, you can build Django microservices applications that meet stringent security requirements.

Summary

In this chapter, we learned how to secure Django microservices by implementing north-south security to control client access to services and east-west security to control inter-service access.

We started by exploring north-south and east-west security. Then, we looked at security implementation techniques and learned about JWT as a token-based technique and OAuth 2.0 as a user-based technique. Finally, we built north-south and east-west security into our sample microservices application with JWT.

With this, you know how to secure microservices if application requirements require it. Furthermore, you have gained enough knowledge to implement OAuth 2.0 security if necessary.

In the next chapter, we'll explore and apply **caching**, which helps increase and maintain application performance.

10

Improving Microservices Performance with Caching

Because microservices run asynchronously, their execution time and performance are of minor importance in most scenarios. For example, when a Django web app offloads the task of sending a confirmation email to the user, it doesn't matter whether the email arrives in 5 or 15 seconds. Nevertheless, we developers want to utilize resources optimally, and in extreme cases, we can end up with microservices taking half an hour or longer to run.

To address this, this chapter teaches you how to maintain and improve microservices performance with caching. First, you'll learn about the fundamentals and benefits of caching. Then, you'll learn about applying Django's cache framework. Finally, you'll master Redis for caching.

By the end of this chapter, you will know how to apply caching in Django microservices applications to maintain or improve performance.

To achieve this, this chapter covers the following topics:

- Introducing caching
- Applying Django's cache framework
- Using Redis for caching

This chapter helps you become an advanced Django microservices developer. Of course, there is always more to learn, and learning never stops, but with the knowledge from this chapter, you can build the most advanced microservices.

Technical requirements

You can find the code files for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter10`.

Introducing caching

Caching is a technique for temporarily storing and reusing data from a high-speed storage device. A **cache** is the actual data storage, generally in the RAM, to store a data subset temporarily to fulfill data requests. Retrieving data from RAM is much faster than retrieving data over a network connection from a database server. So, caching improves application performance significantly.

Because the cache lives in the RAM, it's temporary by nature, and we set a **Time To Live** (**TTL**) to determine how often we'll refresh the cache.

To compare caching with regular data retrieval, let's examine the steps each option takes in a scenario where an application such as a Django app requests data from a database to display on a web page.

Regular data retrieval collects data from a database and requires these steps:

1. An application running at a workstation queries a database running at a server over the network for specific data.
2. The database executes the query and accesses the hard disks on the database server to fetch the data.
3. The hard disks allocate the data and send it to the database.
4. The database formats the data and sends it over the network to the requesting application.
5. The application (Django app) retrieves the data and shows it in a list on a web page.

On the other hand, caching has the data stored in high-speed storage, and retrieval requires these steps:

1. The first application instance queries the database for data.
2. The database retrieves the data and places it into the cache with a TTL.
3. Subsequent application instances query the database and retrieve the data from the cache as long as the TTL validates.
4. When the TTL expires, the cache is refreshed from the database as the first subsequent application instance requests the data.

The data route for caching is substantially shorter and quicker due to fewer steps and high-speed storage:

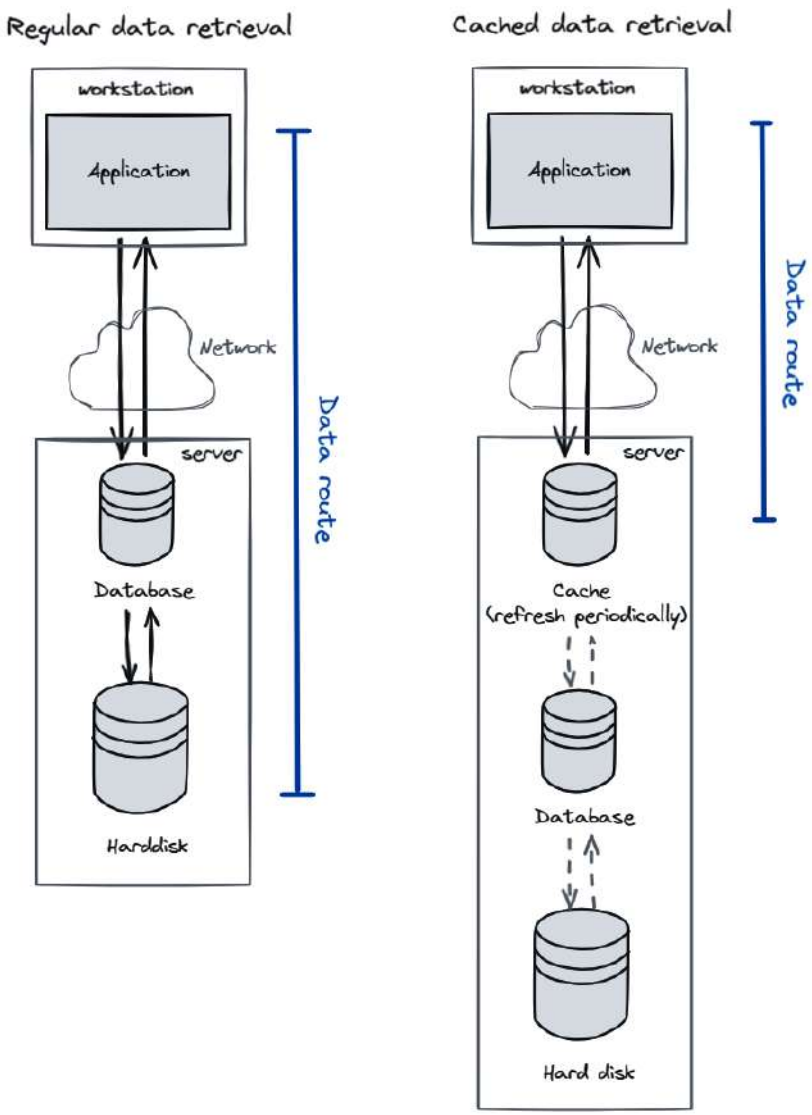

Figure 10.1 – Regular versus cached data retrieval

For caching to work optimally, it's recommended to cache intensively used data that only changes sometimes because the speed benefit occurs mainly when applications request the data frequently. Furthermore, the general advice is to cache data that only changes occasionally because the caching mechanism spends as little of its resources as possible on updating. Even though this is good advice, thanks to today's large amounts of internal memory and fast servers, it can be interesting to cache frequently changing data, as we'll see in the *Using Redis for caching* section.

Regarding implementing caching in Django microservices applications, we have these options:

- **Django's cache framework** to cache Django web pages and data for web pages and microservice workers.
- **Redis caching** to cache data for microservice workers such as our matching address worker. However, because of Redis versatility, it can also serve as the caching backend for Django's cache framework.

In the following section, we'll apply Django's cache framework in our sample microservices application. After that, we'll include Redis caching in our microservice workers.

Applying Django's cache framework

Django's cache framework is a built-in feature to store and retrieve cached data. Django supports several *caching types*, which we can set through the settings in the `settings.py` file. We can set the caching type to one of the following:

Cache type	Description
In-memory	Cache data is stored in internal memory. Pros: • Fast(est) • Suitable for small datasets Cons: • Limited capacity because of available memory • Data is lost when restarting the server
File-based	Cache data is stored in the filesystem. Pros: • Suitable for large datasets • Data persists when restarting the server Cons: • Relatively slow

Database-based	Cache data is stored in the database Pros: • Suitable for large datasets • Data persists when restarting the server Cons: • Relatively slow • Additional database load slows other database operations
Backend-based	Cache data is stored in specialized caching backends such as Redis. Pros: • Suitable for large datasets • Data persists when restarting the Redis server • Scalable through distributing caching Cons: • A bit slower than in-memory caching • Requires a Redis server

Table 10.1 – Django caching types

We'll apply in-memory and backend-based caches because they are the fastest and most applicable to microservice applications. For now, we'll use in-memory caching by adding this section to `settings.py` in the `subscription_celery/ subscription_celery` directory:

```
CACHES = {
    'default': {
        'BACKEND':'django.core.cache.
            backends.locmem.LocMemCache',
        'LOCATION': '<unique-suffix>',
    }
}
```

The `BACKEND` option sets caching to in-memory with the `LocMemcache` value.

The `LOCATION` option identifies individual memory stores and you must fill it with a value of your choice.

Next, we choose the *object to cache*, which can be either of the following:

- A complete Django web page
- Specific data – for example, to display on a web page

We'll examine caching a complete web page and specific data for the **Success** page in our sample microservices application. In fact, the product owner asked whether we could extend the **Success** page by showing the top three magazines because the new subscribers might also be interested in these.

Because the top three don't change often and every subscriber will see the **Success** page, it's a good candidate for caching. First, we'll cache the whole page. Then, we'll only cache the top three pieces of data and let Django read that data from the cache.

Caching a web page

Django's cache framework can cache complete web pages. After caching, Django serves subsequent requests for such pages from the cache instead of regenerating them. Reading from the cache is faster than regenerating. We can cache a page by decorating its definition in `views.py` and specifying a maximum age or TTL for the cache. When the maximum age expires, Django refreshes the page in the cache.

To see how this works, we'll cache the **Success** page. Open the `success.html` file in `subscription_celery/subscription/templates/subscription` and change its content to the following:

```
{% extends 'subscription/base.html' %}
{% block content %}
<h1 name="success_header">Thanks!</h1>
<p>You'll receive the latest edition of our magazine within three
days.</p>
<p>Maybe you're also interested in our other magazines:</p>
<ul>
    <li><b>Django Demistified</b><br/>Go beyond the
        standard Django features and learn features
            only experienced pros know.</li>
    <li><b>Microservices Magic</b><br/>Take yourself to the
        next level as a developer and master
            microservices.</li>
    <li><b>Python to Perfection</b><br/>Squeeze every last
        bit out of Python with the very latest Python
            features, tips and tricks.</li>
</ul>
{% endblock content %}
```

For the sake of showing how caching works and simplicity, we hardcoded the top three magazines as a list.

Now, let's cache this page. Change the `views.py` file to this code:

```
1 import os
2 from django.views.decorators.cache import cache_page
3 from django.utils.decorators import method_decorator
4 from django.views.generic.base import TemplateView
5 from django.views.generic.edit import FormView
6 from dotenv import load_dotenv
7 from subscription.forms import SubscriptionForm
8 from subscription.tasks import match_address_task
9
10 load_dotenv()
11
12 client_token = os.getenv("CLIENT_TOKEN")
13
14 class SubscriptionFormView(FormView):
15    template_name = "subscription/subscription.html"
16    form_class = SubscriptionForm
17    success_url = "/success/"
18
19    def form_valid(self, form):
20        task_message = {
21            "name": form.cleaned_data["name"],
22            "address": form.cleaned_data["address"],
23            "postalcode": form.cleaned_data[
                  "postalcode"],
24            "city": form.cleaned_data["city"],
25            "country": form.cleaned_data["country"],
26            "email": form.cleaned_data["email"],
27          "client_token": client_token
28        }
29        match_address_task.delay(task_message)
30
31        return super().form_valid(form)
32
33 @method_decorator(cache_page(60 * 30), name='dispatch')
34 class SuccessView(TemplateView):
35    template_name = "subscription/success.html"
```

Lines 2–3 import the `cache_page` and `method_decorator` decorators. The `cache_page` decorator marks a page for caching, and we need the `method_decorator` decorator because the **Success** page is a `TemplateView`-based page.

The actual caching takes place in *line 33*, which decorates the `class` definition for the Success page of *lines 34–35*. The `cache_page(60 * 30)` parameter sets the TTL to 30 minutes (60 seconds * 30).

This is all that's needed to cache the **Success** page. Redeploy the sample microservices application and test the subscription app. The **Success** page now shows the top three titles from the cache.

But we can do better than this hardcoded solution. So, let's cache the data of the top three magazines and include this data on the **Success** page in the following subsection.

Caching page data

Instead of hardcoding, we can improve and build a more realistic solution where the data for the top three magazines comes from a database, which we read into the cache. This requires us to do the following steps:

1. Extend the `models.py` with a `Magazine` class to create a `magazine` collection in our MongoDB database.
2. Migrate the updated model to MongoDB.
3. Populate the `magazine` collection.
4. Read the `magazine` collection from MongoDB into the cache.
5. Load the magazine data from the cache into the **Success** page.
6. Show the top three magazines as a list on the **Success** page.

Let's start with *steps 1–3* to extend our model and populate the top three magazines. Add this class to `models.py` in the `subscription_celery/subscription` directory:

```
class Magazine(models.Model):
    title = models.CharField(max_length=100, blank=True)
    description = models.CharField(max_length=500,
                                   blank=True)
```

Then, migrate the updated model to MongoDB. When ready, check the new `subscription_magazine` collection that Django and MongoDB created. It's empty right now. Let's add the top three magazines via the Django shell:

1. Start the Django shell from the `subscription_celery` directory with this command:

```
$ python3 manage.py shell
```

2. Import the `Magazine` model so we can add magazines to the collection:

```
>>> from subscription.models import Magazine
```

3. Define the first magazine to add:

```
>>> mag = Magazine(title='Django Demystified',
           description='Go beyond the standard Django
```

```
            features and learn features only
            experienced pros know.')
```

4. Save the first magazine:

```
>>> mag.save()
```

5. Repeat *steps 3 and 4* for these magazines:

```
title='Microservices Magic'
description='Take yourself to the next level as a
    developer and master microservices.'

title='Python to Perfection'
description='Squeeze every last bit out of Python
    with the very latest Python features,
        tips and tricks.'
```

6. Close the Django shell (`quit()`).

Okay, we populated the top three magazines. Our next step is to read them from MongoDB into the cache, and we can use the `set` method from the `cache` class for this:

```
cache.set("key", data)
```

The `key` argument is an identifier by which we can read the data back from the cache, and `data` can be any regular Python data type such as a string or list.

We only want to load the top three magazines once when we start the Django subscription app. The `apps.py` file is a good place to load the cache because it steers the app configuration at startup. We can add custom functionality to the startup by adding a `ready` method to the `SubscriptionConfig(AppConfig)` class.

To load the top three magazines, change the `apps.py` file in the `subscription_celery/subscription` directory to this content:

```
1 import os
2 from django.apps import AppConfig
3 from django.core.cache import cache
4
5 class SubscriptionConfig(AppConfig):
6    default_auto_field = 'django.db.models.BigAutoField'
7    name = 'subscription'
8
9    def ready(self) -> None:
10       if os.environ.get('RUN_MAIN'):
```

```
11          from .models import Magazine
12          magazines = Magazine.objects.all()
13          cache.set("magazines", magazines)
14          print('Magazines added to Django cache')
```

Line 3 imports the Django `cache` class. *Line 9* defines the `ready` method, which executes on startup. Within the `ready` method, *line 10* ensures that our custom code only runs once. *Line 11* imports the `Magazine` model. We do this import here because the model is only available when the app startup is ready.

Line 12 reads the magazines from the MongoDB collection into the `magazines` variable. *Line 13* writes the magazines to the cache via the `set` command, which takes the `"magazines"` key to identify the data and the actual cache data. We could combine *lines 12* and *13* in one statement, but we split them for clarity.

Next, we must load the cached magazine data into the **Success** page. For this, we need to add a `get_context_data` method to the view definition for the **Success** page in the `views.py` file. The `get_context_data` method enables context data for view-based pages. Furthermore, we can use the `get` method from the `cache` class to read data from the cache like this:

```
cache.get("key")
```

The `key` argument is the identifier of the data in the cache we want to read.

To set the context data for the **Success** page, remove or comment out *line 33* in the `views.py` file to remove the page caching. Then, change the class definition for the **Success** page in the `views.py` file to this definition:

```
1 class SuccessView(TemplateView):
2    template_name = "subscription/success.html"
3
4    def get_context_data(self, **kwargs):
5        context = super().get_context_data(**kwargs)
6        magazines = cache.get("magazines")
7        context['magazines'] = magazines
8        return context
```

Within the `get_context_data` method, *line 5* initializes the `context` variable, which will hold the page's context data. *Line 6* reads the top three magazines from the cache via the `get` command and by specifying the `"magazines"` key.

Then, *line 7* reads the magazines into the `context` variable, and *line 8* loads the context data for the **Success** page.

This leaves us with just one last step: showing the top three magazines as a list on the **Success** page. For this, we change the `success.html` file to the following:

```
{% extends 'subscription/base.html' %}
{% block content %}
<h1 name="success_header">Thanks!</h1>
<p>You'll receive the latest edition of our magazine within
    three days.</p>
<p>Maybe you're also interested in our other magazines:</p>

{% for magazine in magazines %}
<ul>
  <li><b>{{magazine.title}}</b><br/>{{magazine.description}}
    </li>
</ul>
{% endfor %}
{% endblock content %}
```

We have replaced the hardcoded titles with a `for` loop that shows the top three magazines as a list. This finishes our last preparation step in caching page data. Now, redeploy and test the sample microservices application to see the top three magazines when you successfully enter an address.

It's always great to see that everything works. You can even run our automated end-to-end test to check the application. With this, we close our exploration of in-memory caching. In the following section, we'll look at backend caching with Redis, which we already use for task queue management.

Using Redis for caching

Redis is an open source, in-memory data store that provides the following from a Django perspective:

- Message queue brokering for Celery
- Backend caching in collaboration with Django's cache framework
- Standalone caching through the Redis in-memory database

In *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ*, we explored Redis as a message queue broker for Celery. So, we'll now focus on backend and standalone caching with Redis in the following subsections.

Redis as a backend cache for Django's cache framework

Due to its versatility, Redis can also act as the backend for Django's cache framework. The main reason to do so is that Redis backend caching allows large persistent datasets while being only slightly slower than in-memory caching.

Until this point, we used in-memory caching for Django's cache framework, as we set in the *Applying Django's cache framework* section, via the `setting.py` file for the Django subscription app. Likewise, we can set Redis to handle the caching for Django's cache framework.

Change the `CACHES` element in `settings.py` (in the `subscription_celery/subscription_celery` directory) to the following:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.
            redis.RedisCache',
        'LOCATION': 'redis://redis:6379/0',
    }
}
```

The `BACKEND` option sets Redis as the backend cache system. The `LOCATION` option refers to the Redis server running on our system – in this case, our Dockerized Redis server.

> **Important note**
> If you run Redis outside Docker, `LOCATION` needs to be `'redis://localhost:6379/0'`.

This is all it takes to apply Redis as the backend cache for Django. You don't need to change the templates or other Django app parts, so you can redeploy the sample microservices application and test it.

Pretty impressive, but Redis can do even more because it can also run as a standalone cache. This is interesting for our Celery microservices (workers), as we'll see in the following subsection.

Redis as a standalone cache

Redis is also an in-memory database with high-speed access to store data, and that way, Redis can also function as a standalone cache. This feature is interesting for microservices, such as our matching addresses worker, because it can speed up performance.

In general, as microservices run asynchronously, their duration time is of little or no concern. Nevertheless, we developers want to utilize resources optimally. If we have a microservice that applies machine learning or creates an extensive report, it can take a long time to run. Caching can then significantly speed up the execution time.

The same applies to our matching address microservice. Currently, the address collection will probably contain about 10 or 20 addresses, depending on how often you test.

With such a small number, performance is no issue. But what if the address collection contains hundreds of thousands of addresses? In that case, collecting all the base addresses to match from the RESTful API for every match can become lengthy.

So, let's overcome that now and apply Redis to store all the base addresses in Redis' in-memory cache. This requires the following steps:

1. Load the base addresses into the Redis in-memory database. We'll use the run-one-time mechanism of `apps.py` in the RESTful API for this since the RESTful API is responsible for the data layer in our sample application.
2. In the match address worker, read the base address from Redis into the Python list that the worker uses to match a new address.
3. In the match address worker, add a new address to the Redis cache for subsequent matches.

We'll start with loading the base addresses into Redis, and we'll use the `lpush` method from the `redis` class for this:

```
redis.lpush('list_name', listitem)
```

The `lpush` method creates a list with the `list_name` and `listitem` elements we supply. We apply this to our sample application by loading the base address once into the Redis cache via the `apps.py` file from the RESTful API. We'll use the `pymongo` package to read the address collection directly from our MongoDB database.

To do so, change the `apps.py` file in the `subscription_apis/address_api` directory to the following to read the list of addresses into the cache:

```
1 import os
2 import redis
3 from django.apps import AppConfig
4 from pymongo.mongo_client import MongoClient
5 from pymongo.server_api import ServerApi
6
7 class AddressApiConfig(AppConfig):
8    default_auto_field = 'django.db.models.BigAutoField'
9    name = 'address_api'
10
11    def ready(self) -> None:
12      if os.environ.get('RUN_MAIN'):
13          redis_client = redis.Redis(host='redis',
                                       port=6379)
15
16          con = "mongodb+srv://django-microservice:
                <password>@<cluster>/?retryWrites=
                    true&w=majority"
17          client = MongoClient(con, server_api=
                            ServerApi('1'))
18          db = client["Subscription"]
```

```
19          address_col = db["address_api_address"]
20
21          redis_client.delete('addresses')
22
23          for address in address_col.find():
24              redis_client.lpush('addresses',
                                   address["address"])
25
26          print('Addresses added to Redis cache')
```

You already know the code for `pymongo` from *Chapter 4*, *Cloud-Native Data Processing with MongoDB*. So, we'll focus on the code for Redis.

Line 2 imports the `redis` class and enables us to use methods such as `lpush`. *Line 13* creates a Redis client that connects to the Redis server. *Line 21* deletes a possible earlier or old cached list if we restart the RESTful API. *Line 24* loops over the `address` collection and pushes each address into the cached list (named `addresses`) with the base addresses.

Okay, this loads the base address into the Redis cache. Next, we'll read these addresses into the matching list in our match worker, and we'll use the `lrange` method from the `redis` class for this:

```
redis.lrange('list_name', start_element, stop_element)
```

The `lrange` method returns the values from a list from `start_element` to `stop_element`.

To apply this to the match address worker, open the `tasks.py` file in the `subscription_celery/subscription` directory and change the `import` section like this to import the `redis` class:

```
1 import jwt
2 import os
3 import redis
4 import requests
```

Next, add the following line to connect to the Redis server:

```
11 redis_client = redis.Redis(host='redis', port=6379,
                  db=0)
```

Then, replace *lines 40–41* with this line that applies the `lrange` method to read the base address from the cache into the addresses list:

```
39 if calling_producer == expected_client_producer:
40     addresses = redis_client.lrange('addresses', 0, -1)
41
42     top_score = 0
```

Line 40 reads the addresses from the cache. We set `start_element` to `0` to read from the first address and `stop_element` to `-1` to read up to the last available address in the cache.

This reads the addresses much faster than directly from the RESTful API and improves the match address worker's performance. From here, the match address worker uses the cached address list to match like before.

There's only one change left that we need to make: adding a new address to the cache for subsequent matches. We must do this because we no longer read the addresses from the RESTful API, which only refreshes the cached address list when it starts or restarts. So, to refresh and complete the cached address list, we add this line (`65`) to push the new address to the cache:

```
64 response = requests.post(...
65 redis_client.lpush('streets', match_address)
```

Finally, redeploy the sample microservices application and run some tests to try out this cached version.

Thanks to caching, we now have a microservices application prepared for heavy user and data loads. You are now able to develop the most advanced Django microservices applications.

Summary

In this chapter, we learned how to use caching to maintain and improve the performance of Django microservices applications. After exploring caching, we utilized Django's cache framework to cache Django web pages. We ended with applying Redis' in-memory database as a cache for microservice workers to improve their performance as well.

This allows you to develop and deploy advanced microservices applications that perform optimally even as the number of users or data volumes to process increase.

In the next chapter, we'll look at best practices for developing and deploying Django microservices.

11

Best Practices for Microservices

Microservices are a relatively recent development, but fortunately, there is already enough experience to look at best practices. With this, we can profit from earlier hands-on experience and avoid pitfalls to optimize microservices development.

In this chapter, we'll examine best practices for Django microservices. By the end, you'll know how to organize your code, document it, and control its versioning.

To achieve this, this chapter covers the following topics:

- Organizing code
- Documenting microservices
- Logging and monitoring
- Error handling
- Versioning microservices

Some best practices for error handling and monitoring have been addressed before in *Chapter 5*, *Creating RESTful APIs for Microservices*, and *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ*, yet we mention them here to group them in a complete chapter for easy access.

Technical requirements

One of the best practices for microservices concerns version control, and Git is an excellent tool for managing updates to and releasing microservices. If you want to apply Git, you can install it from `https://git-scm.com/downloads`.

You can find the code files for this chapter on GitHub at `https://github.com/PacktPublishing/Hands-on-Microservices-with-Django/tree/main/Chapter11`.

Organizing code

Organizing our microservices code is important for maintainability and collaboration with other developers. Depending on your needs and situation, consider applying the following best practices for organizing microservices code:

- Apply the singe task principle
- Separate responsibilities
- Standardize the communication protocols
- Containerize microservices
- Apply version control
- Document the code
- Conduct code reviews

Now, let's look further into these best practices to organize our code.

Apply the singe task principle

Develop each microservice to execute a single task. This helps keep the codebase focused and makes it easier to understand, test, and maintain. Both for fellow developers and ourselves when we need to change a microservice after a while.

For example, our match address worker is only concerned with matching a new address to base addresses. And the email worker only sends a confirmation email.

Separate responsibilities

Design microservices to separate the following functionality into components like functions and classes:

- **Data processing** for reading and updating data from sources like databases and files.
- **Business logic** for executing the actual functionality, like calculating a total or selecting sales figures for a specific month.
- **External service access** for offloading secondary tasks to other microservices, like sending a confirmation email.

Such a separation improves modularity, which makes it easier and less risky to change a component because it's isolated from other components.

For example, the `match_address_task` function in `tasks.py` now contains all functionality integrally because it contains every step for:

- Data processing: reading the base addresses into a list and writing a new address to the database and cache.
- Business logic: matching the new address to the base address.
- External service access: offloading the task to send the confirmation email.

This is fine since `match_address_task` is a small and orderly function. But if it would span dozens of lines, it would be better to divide the data processing functionality into separate functions like these:

```
def read_addresses():
def create_address(address):
```

The business logic for address matching stays in `match_address_task` because that's its core and, therefore, a single task. Offloading the email sending also remains unchanged because it only spans one line, so separating this into a function won't improve the modularity.

Standardize the communication protocols

Apply uniform communication protocols for:

- Offloading tasks from producers to workers and workers among each other.
- Processing data by workers.

This simplifies integration and improves the reusability of microservices in other applications.

For example, all the producers and workers in our sample microservices application use the Redis protocol to offload tasks. And the match address worker utilizes the protocol for RESTful APIs to process data.

Containerize microservices

Containerize microservices because this:

- Ensures consistency across different environments.
- Improves the scalability.
- Simplifies the deployment.

For example, we dockerized the Django subscription app, the Celery workers, and the RESTful API. This way, we can deploy a new version of our application with a single script. And we can move our application relatively simply to AWS or Azure.

Apply version control

Use a version control system like Git because it gives you control over changes to your code and enables you to track them. As a Django developer, you are most likely familiar with Git. Otherwise, you can check the Git documentation at

`https://git-scm.com/book/n/v2`.

For example, create separate Git repositories for:

- The Django subscription app, including the Celery workers.
- The RESTful API.

This way, we can independently develop and deploy the Django app and the RESTful API. In the subsection, *Versioning microservices*, we go into versioning microservices specifically.

Document the code

Create clear and comprehensive documentation for your code. This helps other developers understand our code and helps us when we change our code later.

For example, create a `README` file that explains the purpose and usage of our software with sections like:

- Introduction
- Features
- Installation
- Configuration
- Usage

In the subsection, *Documenting microservices*, we go deeper into documenting microservices.

Conduct code reviews

Conduct peer code reviews because it improves code quality and adherence to coding standards.

For example, a fellow developer reviewed our code and suggested adding code comments to explain what each function and class does.

Okay, so far, for organizing the microservices code. If you apply these best practices, you lay a good foundation for well-organized code and associated benefits. Next, we'll examine the best practices for documenting our microservices.

Documenting microservices

Documenting microservices is essential because it helps:

- Fellow developers understand our code when they need to modify it or provide third-line support.
- Us when we later change our code.
- System administrators install and configure our microservices application.
- Users to work with our application.

Depending on the requirements, documentation can be extensive and include a complete set of how-to guides and a reference. But at least make sure that you cover the following documentation:

- Code comments to help yourself and other developers understand your code.
- A `README` file to help others to install and use your software.
- A RESTful API reference to help others to work with the API.

Let's examine these documentation types into more detail.

Provide code comments

Commenting code helps you remember why you developed your code like you did. And it helps other developers understand your code when they need to update it.

For example, equip each function with a comment section like this for the `send_email_task` function:

```
@shared_task
def send_email_task(name: str, street: str, email: str,
                    token: str) -> None:
    """
        This function is a Celery worker that sends (upon a
        task request) a confirmation email to the provided
        receiver.
        To send the mail, it utilizes the Django send_mail
        wrapper from the django.core.mail module.

        Parameters:
            name (string): name of the receiver
            street (string): the address determined
            email (string): email of the receiver
            token (string): JWT token to authorize the
            calling producer
        Globals:
```

```
            Not applicable

        Returns:
            None
    """
...
```

Furthermore, provide in-line comments wherever you think it improves the understanding of the code.

Create a README file for the microservices application

A `README` file helps other developers, system administrators, and users to install and use your software. It explains the purpose and usage of your software, and you should at least provide these sections:

- **Introduction**: Give a concise description of your application and its purpose.
- **Features**: Provide a list of your application's features.
- **Installation**: For example, provide an installation script.
- **Configuration**: For example, specific settings in `settings.py`.
- **Usage**: Provide stepwise instructions to get started with your software. Additionally, for microservices, specify the layout of the task message to call the microservice in a format like this for the match address worker:

```
Task message layout match_address_task:

task_message = {
    "name": <string - max 120>,
    "address": <string - max 120>,
    "postalcode": <string - max 15>,
    "city": <string - max 120>,
    "country": <string - max 80>,
    "email": <string - max 120>,
    "client_token": <string - JWT token>,
}
```

Also consider these sections to fully cover your microservices application:

- **Dependencies**: For example, other systems (e.g., Redis) and Python packages.
- **Applied protocols**: For example, the **Advanced Message Queuing Protocol** (**AMQP**) for RabbitMQ.

Also, feel free to add your sections as you'd like and think they are helpful.

A `README` file can have any format you like, but creating them as a Markdown or simple text file is common because all operating system platform support these formats.

If you want to learn more about Markdown, check out its documentation at `https://www.markdownguide.org/`.

Document RESTful APIs

If the RESTful API is also your responsibility, you must document it as well to help other parties to access the API.

API documentation is a field in itself, and you can find an explanation and approach at `https://idratherbewriting.com/learnapidoc/`. But in any case, make sure that you provide an overview of:

- Endpoints and methods.
- Parameters.
- Examples of a request and response for each method.

For instance, provide a `cURL` example for retrieving a list of all base addresses:

```
$ curl -X GET https://127.0.0.1:7000/api/v1/addresses/
```

And present an example of how the response for this command would look like.

This ends the documentation section. If you apply these best practices, you help fellow developers and yourself to maintain your microservice applications efficiently. Furthermore, you allow the users of your application to get started quickly. Next, we'll look into the best practices for logging and monitoring.

Logging and monitoring

Logging microservices can help track errors and failures. **Monitoring** helps optimize microservices and can signal exceptional occurrences like security attacks.

In *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ*, we already examined monitoring Celery with the `Flower` package and RabbitMQ with its built-in features. Here, we focus more on signaling exceptions and reacting to such exceptions.

Depending on your situation and requirements, you can apply the following best practices for logging and monitoring:

- Apply integrated logging
- Implement log levels
- Alert anomalies

Next, we'll look further into these best practices.

Apply integrated logging

You can build your own logging system, but applying the standard `logging` module for Python is more efficient because it provides a comprehensive set of logging features.

For example, we can log the send mail error in `send_email_task` to a text file instead of printing it to the console:

```
1 import logging
   ...
9 logging.basicConfig(filename="workers_logs.txt",
                      filemode="w",
                      format="%(asctime)s → %(name)s → %
                      (levelname)s: %(message)s")
   ...
89 @shared_task
88 def send_email_task(name: str, street: str, email: str,
                       token: str) -> None:
89     calling_producer = decode_token(token, 'service')
                          ['service_producer']
90     if calling_producer == expected_service_producer:
91         send_mail(
               "Your subscription",
               f"Dear {name},\n\nThanks for subscribing
               to our magazine!\n\nWe registered the
               subscription at this address:\n{street}.\n\n
               And you'll receive the latest edition of our
               magazine within three days.\n\n
               CM Publishers",
               "magazine@cm-publishers.com",
               [email],
               fail_silently=False,
          )
```

```
92     else:
93         logging.error("Authentication failed")
```

Line 1 imports the `logging` package and *line 9* configurates the logging to write log messages to the `workers_logs.txt` text file. And sets the format for the error message.

Line 93 writes an error message to log file if the token authentication fails. For example, a message like this:

```
2024-02-15 09:13:48,975 → root → ERROR: Authentication failed
```

For more information on Python logging, check this tutorial at `https://docs.python.org/3/howto/logging.html#logging-basic-tutorial`. This will help you to further apply logging.

Implement log levels

Implement appropriate log levels to differentiate between different log message types. A common way to classify log messages is this breakdown:

- **INFO**: Use this type to log messages that do not require action, such as the start and end times of a worker's execution.
- **WARNING**: Use this type to log messages that might require action, like a retry count for accessing a RESTful API, which could imply an unstable network connection.
- **ERROR**: Use this type to log messages that require action, such as an unreachable RESTful API.

By identifying log messages like this, you can quickly spot messages that require action. Furthermore, the log levels enable us to implement automated actions on specific log levels.

For example, we could have a Python monitoring script running that watches for `ERROR`-level messages in the log file and sends an email to people for further action.

Log context information

To review logs quickly and efficiently, it's helpful to add context information to log messages like:

- The program, script, task, function, or class that logs the message.
- Date and time of logging.
- The requesting User ID, if applicable.
- Entity identifiers such as order numbers, invoice numbers, employee IDs, Customer ID, or Product ID.
- Error messages caught by the program, script, or task that logs the message.

Such context information can also be interesting to pass along in automated log actions.

Alert anomalies

When you automate handling `Error`-level messages, you can instantly send text or WhatsApp alerts to inform people. Or even shut down your application.

For example, if an automated monitoring process signals a security attack, it can text system administrators and shut down the application to prevent further harm.

By applying these best practices for logging and monitoring, you optimize microservices execution. Next, we'll look into the best practices for error handling regarding microservices.

Error handling

Error handling is essential for developing robust and resilient microservices.

Depending on the robustness requirements for your microservice application, error handling can vary from simply catching and logging errors to extensive retry mechanisms or gentle degradation.

Catching and logging errors should be the minimum for our microservices, so we'll look specifically at that.

Catch and log errors

Python has the `try...except` block structure to catch errors and unexpected events and it's good practice to incorporate this into your workers to make them robust and resilient. Moreover, if we log errors from `try...except` blocks, we combine logging and error handling, which further optimizes microservice execution.

For example, we can extend `send_email_task` to handle errors like this:

```
1 import logging
2 from smtplib import SMTPException
   ...
9 logging.basicConfig(filename="workers_logs.txt",
                      filemode="w",
                      format="%(asctime)s → %(name)s → %
                      (levelname)s: %(message)s")
  ...
89 @shared_task
88 def send_email_task(name: str, street: str, email: str,
                       token: str) -> None:
89     try:
90         calling_producer = decode_token(
```

```
                                token, 'service')
                                ['service_producer']
91          if calling_producer ==
            expected_service_producer:
92              send_mail(
                    "Your subscription",
                    f"Dear {name},\n\nThanks for subscribing
                    to our magazine!\n\nWe registered the
                    subscription at this address:\n{street}.
                    \n\nAnd you'll receive the latest
                    edition of our magazine within three
                    days.
                    \n\nCM Publishers",
                    "magazine@cm-publishers.com",
                    [email],
                    fail_silently=False,
              )
93          else:
94              logging.error("Authentication failed")
95      except SMTPException as e:
96          logging.error(f"Sending email failed with this
                        error: {e}")
97      except:
98          logging.error(f"Sending email failed.")
```

Line 2 imports the exceptions for `smtplib`, which address possible errors when sending emails. `smtplib` refers to the **Simple Mail Transfer Protocol** (**SMTP**), which is a standard protocol to transmit emails between mail servers.

Line 89 starts the `try...except` block, where *lines 95-96* catch errors related to SMTP and write a log message to the log file if an error occurs. Likewise, *lines 97-98* catch all other errors and write log messages if they occur.

Other error handling options

Depending on your robustness requirements, you should consider the following other error handling options:

- Implement a retry mechanism to handle temporary issues without immediately returning an error.

  For example, when our match address worker still retrieved the base address from the RESTful API, we could have built a retry mechanism if the RESTful API didn't respond. And only return an error if the RESTful API doesn't respond within five retries.

- Incorporate gentle degradation in case a dependent service is unavailable.

  For example, we could develop functionality for the match address worker that writes task messages to a temporary data store if the RESTful API or the send mail worker is unavailable. And then when the service is available again, still process the stored task messages.

With these best practices for error handling, we have multiple options to fulfill the robustness requirements for our microservices applications. Next, we'll look at the best practices for versioning microservices.

Versioning microservices

Versioning microservices is vital to ensure the controlled evolution of our microservices, and these best practices help us with that:

- Apply semantic versioning.
- Utilize RESTful API versioning.

Let's further examine these best practices.

Apply semantic versioning

Apply semantic versioning for your microservices to express the meaning of the changes in a specific version. Semantic versioning follows the `major.minor.patch` structure for version numbers. Because of this, the version number tells how extensive a new version is.

For example, a version number like `2.0.0` indicates a major release with significant changes and potential impact. While a version number like `4.1.21` refers to a small patch with most likely little impact.

We could setup complete versioning for our Django microservices application (project) as described at `https://packaging.python.org/en/latest/guides/distributing-packages-using-setuptools/`.

But if we do not distribute our project publicly, we can also suffice with adding a version number like this to the `tasks.py` file:

```
1 __version__ = "1.2.9"
2 ...
```

This sets the dunder (double underscore) global `__version__`, which tells us that this version of our Celery workers concerns a patch.

Utilize RESTful API versioning

It's also good practice to provide versioning for RESTful APIs because then requesting parties know what version-specific endpoints and methods they must use (as described in your RESTful API documentation).

In general, RESTful API versioning is path-based, meaning the version is part of the endpoints like this:

```
http://address-api:7000/api/v1/addresses/
```

It's common to have whole-digit version numbers only for RESTful APIs. But you can deviate from that.

For example, we defined this version for our RESTful API in its urls.py file like this:

```
1 from django.urls import include, path
2 from rest_framework import routers
3 from .views import AddressViewSet
4
5 ver = 'v1'
6
7 router = routers.DefaultRouter()
8 router.register(f'api/{ver}/addresses', AddressViewSet)
9
10 urlpatterns = [
       path('', include(router.urls)),
   ]
```

Line 5 sets the `ver` variable to the version number and *line* 8 exposes the version number as part of the endpoint paths.

Okay, this finishes the best practices for microservices. They'll help you to profit from earlier hands-on experience and avoid pitfalls. However, best practices continuously evolve, so checking Django and Python forums for new insights is always wise.

Summary

In this chapter, we examined the best practices for developing and running django microservices applications. We started with how to organize our code optimally. Then, we looked into documenting microservices. Next, we explored logging, monitoring, and error handling for microservices. Finally, we addressed versioning microservices.

With this, you can build even more advanced, robust, resilient, and maintainable microservices.

In the next and final chapter, we'll learn how to transform an existing monolithic Django application into a microservices version.

Further reading

The best practices for Django microservices will keep evolving, and the Internet is a great place to discover new best practices at sites like:

- `https://www.reddit.com/r/django/`
- `https://stackoverflow.com/questions/tagged/django`

12

Transforming a Monolithic Web Application into a Microservices Version

Now that you know the possibilities of microservices, you probably have existing monolithic web applications that you would like to rebuild to better meet user and stakeholder requirements. For example, that reporting application where the user has to wait minutes until a PDF report is ready. Or that credit check application that consults an external **Application Programming Interface** (**API**) for credit data that makes the user have to wait minutes before he can continue.

Such applications are good candidates for rebuilding. Fortunately, rebuilding is a relative term here and amounts more to transformation, where you reuse the existing code as much as possible.

In this chapter, we'll explore an approach for transforming a monolithic web application into a microservices version. By the end, you'll know how to decompose a monolithic web application and design it into a microservices version. You'll also know the follow-up steps leading to testing and deploying the microservices version.

To cover this, this chapter addresses the following topics:

- Introducing the transformation approach
- Implementing the approach step by step

This is the final chapter, where it all comes together. We'll take the Discount Claim app from *Chapter 1*, *What Is A Microservice?*, as an example to transform. But I suggest you simultaneously take an example Django application from your practice and do at least the decomposition and design for this application. That way, it will best fit your practice, allowing you to get the most out of it.

Introducing the transformation approach

The steps of the transformation approach are very similar to the steps of standard development methods, which makes sense because both cases involve application development. So, as with standard development methods, we start with determining requirements and end with deploying the application:

1. Determining requirements
2. Decomposing the monolith
3. Designing the microservices
4. Selecting the technology
5. Creating the data foundation
6. Developing the microservices
7. Testing and deploying

Actually, the decomposition and development steps are the only ones that are substantially different. The decomposition step because we assume an existing application. And the development step because we re-use existing code as much as possible.

This stepwise presentation might suggest a waterfall approach where we perform each step after the other. That's fine if that suits your situation, but it certainly doesn't have to, and we can also do the steps just fine according to Scrum in sprints or any other agile approach.

In the following subsection, we'll look at each transformation step for the Discount Claim app and the application you chose from your practice.

Implementing the approach step by step

Finding candidate monolithic Django applications to transform is relatively simple: Look for applications requiring users to wait until one or more steps are completed and where the steps are independent. The Discount Claim app is such an application because the user experience freezes while the app sends a Slack message and confirmation email and writes to the log:

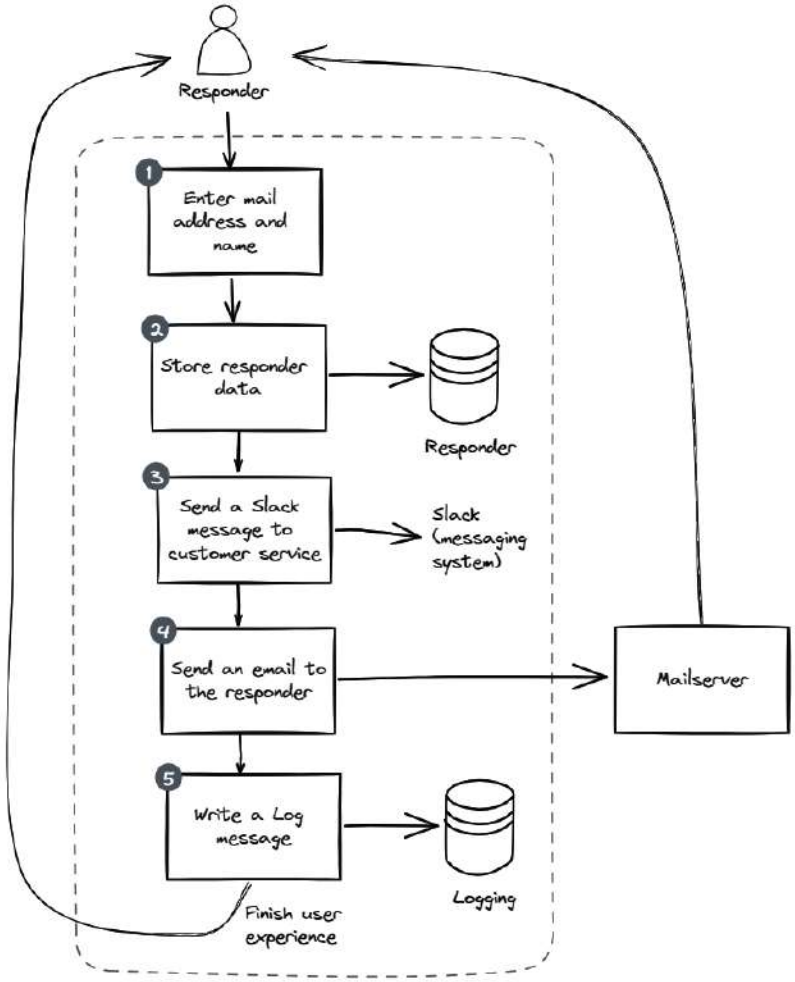

Figure 12.1 – The Discount Claim monolithic app with synchronous tasks

Now, look for such an application in your practice, and if you find one, start determining its requirements as we'll look over next.

Determining requirements

Since we're going to rebuild an existing application, this is an ideal moment to ensure the requirements for that application are still valid. Maybe requirements changed over time due to customer experience or business developments. So, meet up with stakeholders, users, and the Product Owner. And draw up the changed and new requirements.

The *functional requirements* for the Discount Claim app are unchanged, as this user story shows:

As an early bird responder, I want to claim a discount code for the new AI-powered code editor so that I can buy the code editor at a lower price.

Regarding the *technical requirements*, the Product Owner does have a new requirement to improve the user experience:

As the provider of the claim app, I want the app to go directly to the success page after a responder claims a discount without waiting for the confirmation email to be sent.

This requirement fits seamlessly with a microservices architecture, where sending email can become an asynchronous task.

> **Determine the requirements for your application**
>
> Now, you look for new requirements regarding your own application and record them as user stories.

With the existing and new requirements as a starting point, our next step is to decompose the existing monolithic application.

Decomposing the monolith

Microservices are independent and single tasked pieces of software. So, once we have selected a transformable monolithic application, we need to identify the parts we can isolate or decompose as independent microservices.

Pragmatically, the first place to look for decomposable components in a Django application is the `views.py` file because this is generally where the main functional processing occurs within a Django app.

So, within the `views.py` file, we look for distinct functionalities that can run isolated and we notice the `ClaimFormView` class that drives the web page where responders drop their name and email address to claim their discount:

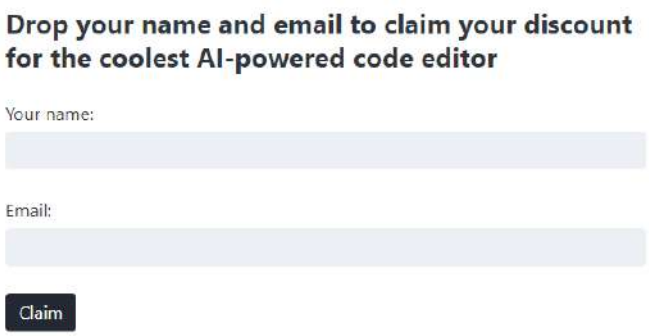

Figure 12.2 – The Discount Claim page

Within the `ClaimFormView` class, there's the `form_valid` method, which executes when a responder clicks the **Claim** button. And the `form_valid` method executes all the claim processing tasks sequentially:

```
class ClaimFormView(FormView):
...
    def form_valid(self, form):
        ...
        # Store the responder data
        ...
        # Send a Slack message to CS
        ...
        # Send a confirmation email to the responder
        ...
        # Write a log message
        ...
    return super().form_valid(form)
```

Only when the last task is finished does the user experience proceed, which can cause long waiting times.

Let's review each task and determine if it's a distinct functionality that can run isolated:

1. The first task stores the responder data into a database. This task has no dependency on one of the other tasks, so we can isolate it as a microservice.
2. The second task sends a Slack message and also has no dependencies. So, we can isolate this task as a microservice as well. Great.
3. The same goes for the task that sends the confirmation email. So, we can decompose a third microservice. It's getting better and better.

4. What about the last task? Can we also isolate the log message writing? Absolutely, so we end up with four decomposed microservices like in this overview:

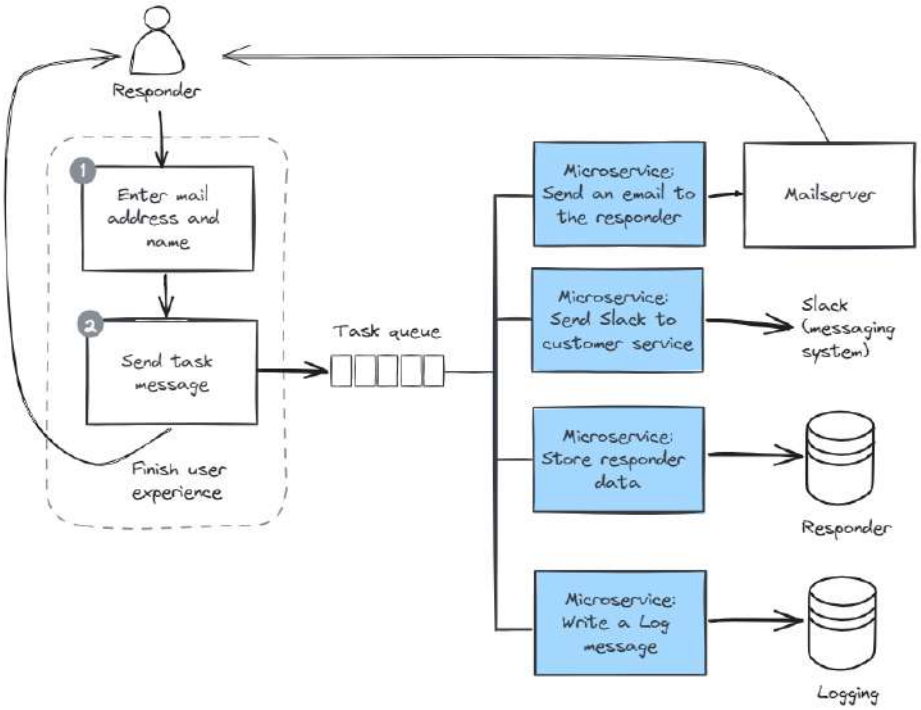

Figure 12.3 – The Discount Claim microservices app with asynchronous tasks

Now, when the responder clicks **Claim**, the user experience proceeds immediately while the microservices run asynchronously.

In summary, we decompose this list of microservices:

- Send email
- Send Slack message
- Store responder data
- Write log message

All of these perform a single task that can run asynchronously.

Decompose your application

It's your turn again. Identify and isolate the distinct functions in your application and mark them as microservices.

With this decomposition, we move over to the next step, where we design the decomposed microservices.

Designing the microservices

Having identified and isolated the microservices within the monolith, it's time to design what the microservices exactly must do. With the single-task principle in mind, we design and specify each microservice in terms of:

- What it does. Defined as a list of the main steps.
- Who (in system terms) implements the microservice's functionality. For example, a Python program or a Celery worker.
- What the microservice triggers to run. For example, a client producer or another microservice.
- What the outcome of microservice is.

Such a design serves the following purposes:

- Ensure that the microservices is single tasked based on the steps and outcome.
- Validate that we interpreted the requirements correctly.
- Act as a blueprint for developing the microservice.
- Act as a blueprint for deriving test cases.

There are various techniques for design. We use the *use case* technique because it specifies triggers and actors, which coheres nicely with the setup and nature of microservices. Furthermore, use cases are also easy to follow for non-development colleagues.

If we apply the use case technique for the send email microservice, we come to this use case:

Overview	Send an email with a discount code.
Actor	A microservice written in Python.
Trigger	A new task arrives in the task queue.
Steps	Parse the task. Prepare an email. Open a connection to the mail server. Send the email request to the mail server.
Outcome	The responder/user receives the email.

Table 12.1 – Send email use case

This specifies the microservice clearly and concisely, which is just what we want. To keep this subsection concise, we omit the other use cases because their design is identical to the send email use case.

Design the microservices for your application

Now, you draw up the use cases for the isolated microservices in your application. Do they have a single outcome? Do the steps perform only one task?

If we finish designing our microservices, we'll take the next step and look at what technology best fits our microservices application.

Selecting the technology

The design step mainly focuses on the what of our microservices, and next, this step focuses on the *how* in terms of the technology to implement and execute the microservices.

For that, we answer these questions for each microservice:

- Do we need to containerize the microservice because it needs to run on different platforms or be scalable? In our case, we select Docker for containerization.
- Must the microservice run locally or in the cloud? We choose locally, but otherwise, we could have selected AWS or Azure, for example.
- Does the microservice process data in a database or other data source? And do other microservices or applications need the same data processing? If yes, build an API microservice. In our case, we select the RESTful API architecture for the microservice that stores the responder data.

- What's the best and most appropriate data storage for the microservice? We could select MongoDB, but because the microservice runs locally and we have PostgreSQL already in-house, we chose this database. Furthermore, PostgreSQL also runs as a Docker container, which fits well with our setup.
- What message queue broker and task management system do we apply for the microservice? We select Redis as the message broker and Celery as the task manager because they are common for Django microservices.
- Does the microservice profit from caching? Since we already have Redis as the message broker, we'll also apply it for the caching.

This list of questions covers most technical aspects of microservices but isn't exhaustive. So, consider other technical matters if the requirements demand to do so.

Select the technology for your application

Now, answer the technology questions for the microservices in your application. What message broker did you choose? Do you foresee the need for caching?

After selecting the technology, we have finished the preparation and are ready to go to the implementation steps. In doing so, we start with the data foundation for the microservices.

Creating the data foundation

Like any other program or script, microservices process data in one way or another. Each microservice receives or retrieves data, processes it, and outputs any data. As a result, data is fundamental to microservices, and it's logical to create the data foundation first.

The steps to create the data foundation depend on the nature of our microservices and the technology choices we made, but come down to this:

1. Create the database, including the tables, stored procedures, and other required components like triggers or user-defined functions. We create a database called `DiscountClaims` with tables like `responder` and `logging`.
2. Create the required APIs. We develop a RESTful API for storing the responder's data in the `DiscountClaims` database.

If you have chosen to apply a cloud database like Azure SQL, you should add a step for selecting and activating the corresponding subscriber model.

Create the data foundation for your application

Now, you create the data sources for your application. Does it involve a RESTful API? If so, you've also created your first microservice for your application.

With the data in place, we're ready for the next step, which is developing the microservices.

Developing the microservices

As developers, we're really in familiar territory now and apply our expertise and all the knowledge gathered so far. To transform a monolithic Django app, this comes down to building the microservices with the selected technology stack and reusing the existing code as much as possible:

1. Copy and embed the task-related code from `views.py` (or other source files) into the microservice workers.
2. Replace the copied task-related code by task calls.

For the Discount claim application, we develop the microservices as Celery workers as we did in *Chapter 6*, *Orchestrating Microservices With Celery and RabbitMQ*. For example, we create a `tasks.py` file where we copy and embed the task-related code into:

```
...
@shared_task
def send_email(email_msg):
    ...

@shared_task
def send_slack_message(slack_msg):
    ...

@shared_task
def store_responder(responder):
    ...

@shared_task
def write_log(log_msg):
       ...
```

Next, we replace the task-related code in `views.py` by these task (function) calls:

```
class ClaimFormView(FormView):
...
    def form_valid(self, form):
        ...
```

```
        send_email(email_msg)
        send_slack_message(slack_msg)
        store_responder(responder)
        write_log(log_msg)
        ...
    return super().form_valid(form)
```

In addition to functionality, at this stage, we also build security measures and caching if required. And develop any additional requirements as well.

Develop the microservices for your application

Now, you split your `views.py` and the other source files into microservice workers. Reuse the existing code as much as possible. Do you also have to fulfill new requirements? Do you implement them as microservices as well?

Once we have developed the microservices, there's only one step left, and that's testing and deploying.

Testing and deploying

Even though we reuse existing code, we must ensure our microservices work as expected and set by the requirements. The same, of course, applies to any new requirements we implement.

To test our microservices, we apply the following testing approaches learned from *Chapter 7*, *Testing Microservices*:

1. Unit testing
2. End-to-end testing

For the Discount Claim application, we create an automated unit test for each microservice that covers the happy path for claiming a discount. We supplement the unit tests with a manual end-to-end test that fills all form fields to their maximum (boundary) size.

Once we have fixed any errors and the tests run flawlessly, we dockerize and deploy the Discount Claim application as in *Chapter 8*, *Deploying Microservices with Docker*.

Test and deploy your application

Now, you set the final step and test and deploy your application. Do you automate your end-to-end test with Selenium? And do you deploy your application via a Shell script?

Way to go; you transformed your first monolith into a microservices version that optimizes the user experience. There is no better way to finish this chapter and book than with such a positive feeling.

More so because you now know and master all that's needed to develop Django microservices. You developed yourself into an advanced Django developer—it's time to update your resume.

Summary

This chapter taught us the steps for transforming an existing monolithic application into a microservices version. We started with determining the requirements and finished with deploying the microservice application.

With this, you master an approach to effectively and efficiently re-engineer your existing Django applications to improve user experience and meet modern application requirements.

Next, it's your turn to put your knowledge into further practice. Enjoy the Django microservices ride and amaze the world with great applications!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

U

V

W

www.packtpub.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 packtpub.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.packtpub.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.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Web Development with Django - Second Edition

Ben Shaw, Saurabh Badhwar, Chris Guest, Bharath Chandra K S

ISBN: 978-1-80323-060-3

- Create a new application and add models to describe your data
- Use views and templates to control behavior and appearance
- Implement access control through authentication and permissions
- Develop practical web forms to add features such as file uploads
- Build a RESTful API and JavaScript code that communicates with it
- Connect to a database such as PostgreSQL

Hands-On Web Scraping with Python - Second Edition

Anish Chapagain

ISBN: 978-1-83763-621-1

- Master web scraping techniques to extract data from real-world websites
- Implement popular web scraping libraries such as requests, lxml, Scrapy, and pyquery
- Develop advanced skills in web scraping, APIs, PDF extraction, regex, and machine learning
- Analyze and visualize data with Pandas and Plotly
- Develop a practical portfolio to demonstrate your web scraping skills
- Understand best practices and ethical concerns in web scraping and data extraction

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.

Share Your Thoughts

Now you've finished *Hands-On Microservices with Django*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83546-852-4`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly