With examples in Python

Microservice APIs

José Haro Peralta

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Microservice APIs
With examples in Python
Version 10

For more information on this and other Manning titles go to
manning.com

welcome

Dear reader,

Thanks for purchasing the MEAP for *Microservices APIs*. I truly appreciate the time and effort that you'll invest in reading this book, and I hope you find the material up to your standards and worthy of your time.

To get the most out of this book, you'll have some experience with web development and ideally are comfortable working with Python. You don't need specific experience with microservices or web APIs. If you have experience in those areas, you'll have probably found that working with microservices and API integrations can sometimes be frustrating. Operating a distributed platform and tracing errors through it can be challenging, if not daunting, if you don't have the right tools under your belt. And surviving first contact between an API and a client is more of an aspiration than a reality in many teams.

The patterns, practices, and recipes that I share in this book with you have been collected over years of experimentation, and often frustration, with building microservices platforms and driving integrations through web APIs. Microservices come in many forms and shapes, but I've found that there are a number of patterns and practices that, when adopted correctly, get you in a very good position to deliver successful results.

You'll learn about service decomposition strategies, the different types of web APIs that you can use to integrate them, the practice of documentation-driven development for successful API integrations, and a collection of useful patterns that will help you drive better integrations and service implementations. You'll also learn recipes useful for microservices deployments and operations.

Throughout this book you'll find numerous concepts and definitions seasoned with code snippets along the way. I've tried to conceive of a book as practical as it can be, a book that can be useful and valuable in your day-to-day activities. But this book can also help you take a step back and form a higher-level overview of how microservices architectures integrate at a higher level.

This MEAP gives you a unique opportunity to contribute to this book and improve it with your comments. I'll certainly appreciate such feedback. May the journey through microservices and APIs be a fruitful one for you!

If you have any questions, comments, or suggestions, please share them in Manning's liveBook Discussion Forum.

- José Haro Peralta

brief contents

1

What are microservice web APIs?

This chapter covers

- **What microservices are and how this architectural pattern differs from monolith applications**
- **What web APIs are and why it makes sense to implement web APIs as microservices**
- **What the Python ecosystem can offer to develop microservice APIs more easily**
- **The most important challenges of developing and managing microservices**

This chapter introduces and defines the most important concepts in this book: microservices and web APIs. Microservices are an architectural style that structures an application as a collection of loosely coupled services, and web APIs are the web interfaces that allow us to interact with those services. We will see the defining features of microservices architecture, and compare them with those of *monolithic* applications. In contrast with microservices architecture, monolithic applications are structured around a single codebase and deployed in a single build. If you have experience working with traditional web development frameworks such as Django or Flask, you are probably already familiar with the features of a monolithic application. In this chapter, we draw examples from those types of frameworks to illustrate the ideas that we describe.

We also explain what web APIs are, why they are so important for modern web applications, and why it makes sense to deploy them as microservices. We'll see how Python provides a rich ecosystem which makes it ideal for the development of API-driven microservices. This ecosystem includes a wide variety of libraries and technologies, as well as support from different vendors that enable microservices deployments.

We will see that microservices are not a silver-bullet against all problems that web applications present. In fact, they introduce new problems of their own. The last part of this chapter talks about some of the most important challenges that we face when designing and implementing microservices, as well as when operating them. This discussion is not to deter you from embracing microservices, but instead so that you can make an informed decision about whether microservices are the right of architecture for you at this time, and if you decide it is, so that you know what to expect. In later chapters, we discuss strategies that can be used to address the problems raised in this section.

1.1 What are microservices?

In this section, we will see the defining features of microservices architecture. In order to illustrate the concepts, we will draw comparisons with the monolithic architecture pattern. We will use a Django application as an example of monolithic architecture, and explain how the same application could be rearchitected using microservices.

1.1.1 Defining microservices

Let's begin by defining what we understand by microservices. Microservices can be defined in many different ways, and, depending on which aspect of microservices architecture we want to emphasize, authors provide slightly different yet related definitions of microservices. Sam Newman, one of the most influential authors in the field of microservices, provides a minimal definition:

> Microservices are small, autonomous services that work together.[1]

This definition emphasizes the fact that microservices are applications that run independently of each other, yet can collaborate in the performance of their tasks. The definition also emphasizes the fact that microservices are "small." In this context, the idea of being "small" doesn't refer to the size of the microservices' codebase, but to the fact that microservices are applications with a narrow and well-defined scope, following the Single Responsibility Principle of doing one thing, and doing it well.

A seminal article written by James Lewis and Martin Fowler provides a more detailed definition along the same lines. Lewis and Fowler define microservices as architectural style with

> an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.[2]

This definition emphasizes the autonomy of the services by stating that they run their own, independent processes. Lewis and Fowler also highlight that microservices have a narrow scope of responsibilities by saying that they are "small", and they explicitly describe how microservices can collaborate or communicate by using lightweight protocols, such as HTTP.

From the previous definitions, we can see that microservices can be considered as an architectural style in which services are designed as components which perform a small and clearly defined set of related functions. This definition means that, as a general rule, a microservice is designed and built around a specific task or domain; for example, processing payments, sending emails, or handling orders from a customer. Such services are deployed as independent processes, typically running in independent environments, and expose their capabilities through well-defined interfaces. In this book, we will be dealing with microservices which expose their capabilities through web APIs, though other types of interfaces are also possible, such as message queues[3]. It's important to realize that microservices are composed of at least two layers: a layer which implements the capabilities of the service, and a layer which implements the interface to the service. Figure 1.1 illustrates this definition of microservices.

[1] Sam Newman, *Building Microservices* (Sebastopol, O'Reilly, 2015), p. 2.
[2] James Lewis and Martin Fowler, "Microservices", https://martinfowler.com/articles/microservices.html [accessed 2nd Aug 2020].
[3] For a comprehensive view of the different interfaces that can be used to enable communication and collaboration among microservices, see Chris Richardson, *Microservices Patterns* (Manning, 2018).

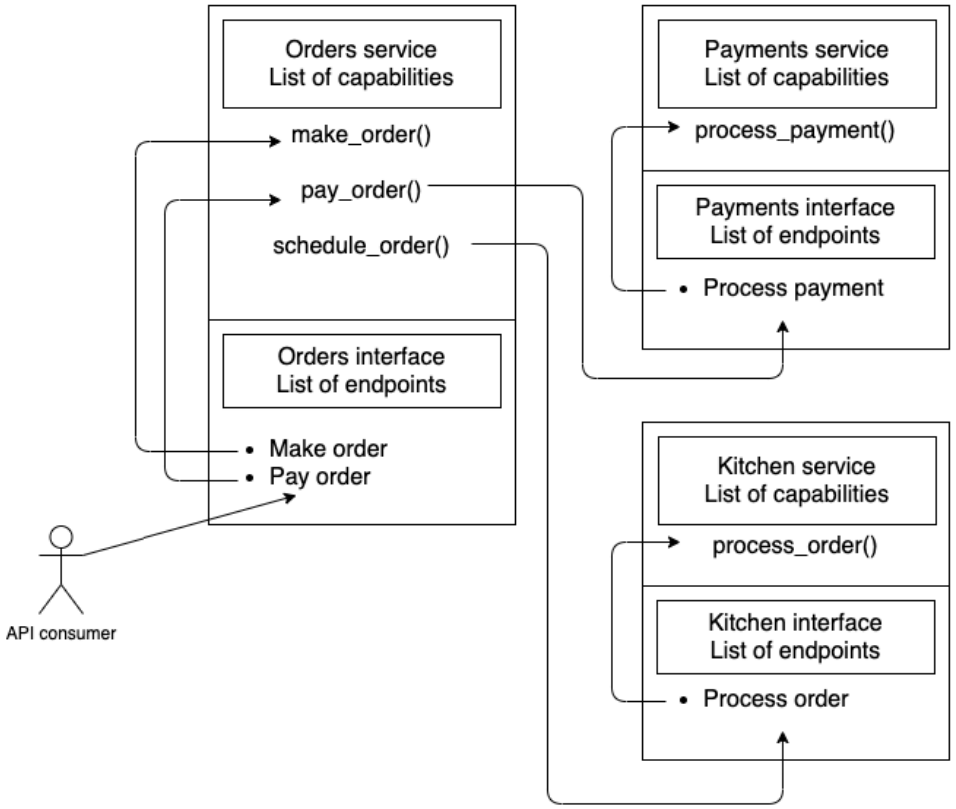


Figure 1.1 Microservices are independent applications which run in different processes and communicate with each other through interfaces which expose selected capabilities of the service. This illustration presents three services: an orders service, a payments service, and a kitchen service. Each of these services implements a number of capabilities and exposes some of those capabilities through an interface.

Figure 1.1 contains a representation of three microservices: an orders service, a payments service, and a kitchen service. The example of capabilities implemented by these services is here simplified for illustration purposes, and each capability is represented by a function. The orders service can take orders from a user as represented by the `make_order()` function. It can also take payments for the orders (`pay_order()`), and schedule an order for preparation (`schedule_order()`). Out of these capabilities, the orders service exposes those which are most relevant for users interacting with the service: the ability to take orders, and the ability to pay for an order. The orders service exposes those capabilities through an interface which defines specific endpoints for each of those actions. We are not being specific at this point about the type of API that the orders service exposes, since there are many types of APIs that microservices can use to expose their capabilities, and we'll discuss them in more detail in Chapter 2.

To fulfil some of its capabilities, the orders service requires the collaboration of other services. For example, as illustrated in figure 1.1, the `pay_order()` capability needs to interact with the payments service to be able to take payments for an order. It does so by accessing the

`process_payment()` capability of the payments service through an API endpoint. Similarly, in order to be able to schedule an order, the orders service needs to access the `process_order()` capability of the kitchen service, which is exposed through the kitchen API.

Now that we have seen what the basic concept behind microservices is, let's bring the concept home by comparing microservices with another type of application architecture which you may be more familiar with: monolithic applications. In order to proceed with such analysis, we'll first define what a monolith is. Then we'll introduce an example that will guide our discussions along this book and will help us illustrate the ideas and concepts that we want to explain about microservices and APIs: the CoffeeMesh application. Finally, we'll compare the features of a monolithic architecture with microservices architecture.

1.1.2 What is a monolith?

This section furthers our discussion of microservices architecture by comparing it with a different type of application architecture which you may be more familiar with: monolithic applications. What exactly is a monolithic application? As Sam Newman explains, in its most fundamental sense, a monolith is a system where all functionality has to be deployed together. Newman distinguishes three types of monolithic architectures: single-process monoliths, distributed monoliths, and third-party black-box systems[4].

For the purposes of our discussion, here we are interested in the single-process monolith. A single-process monolith is an architectural pattern in which the whole application code lives under the same directory and is deployed together as part of a single build. The application running in a monolith may be connected to one or more databases which run in different processes, and possibly in different environments. However, the main logic for the application still lives under a single directory and runs within the same process. In this sense, the fact that the database runs in a different process is accessory. Figure 1.2 contains an illustration of a single-process monolith together with a database. We can run multiple instances of a monolithic application and have them run in parallel for redundancy and scalability purposes. But again, the whole application runs in each of those processes and therefore we can't take advantage of a truly distributed system.

An example of a monolithic application is a website implemented in Django or Flask, two popular web development frameworks for Python. A monolith is not intrinsically good or bad, and it's not necessarily better or worse than microservices. As we'll see, each of these architectures have their advantages and disadvantages, and each has its own use cases. It's typical for many projects to start out as a monolithic Django or Flask application and then evolve into microservices.

Now that we know what a monolith is, let's introduce the example of the CoffeeMesh application and see what its architecture would look like if it was implemented as a monolith and how that would be different from a microservices approach.

1.1.3 Introducing the CoffeeMesh application

This section introduces the CoffeeMesh application, which will serve as a guiding example in this book to illustrate the concepts and ideas that we want to explain about microservices and APIs. CoffeeMesh is a company that allows users to order all sorts of products derived from coffee, including beverages and pastries. CoffeeMesh has one mission: to make and deliver the best coffee in the world on demand to its customers, no matter where they are, no matter when they place

[4] Sam Newman, Monolith to Microservices: Evolutionary Patterns to Transform your Monolith (Sebastopol CA, O'Reilly, 2020), p. 12.

their order. The production factories owned by CoffeeMesh form a dense network, a mesh of coffee production units which spans several countries. Coffee production is fully automated, and deliveries are carried out by an unmanned fleet of drones operating 24/7 all year around.

When a customer places an order through the CoffeeMesh website, the ordered items are produced on demand. An algorithm determines which factory is the most suitable place to produce each item based on available stock, the amount of pending orders which the factory is taking care of (the length of the job queue), and distance to the customer. Once the items are produced, they're immediately dispatched to the customer. It's part of the mission statement of CoffeeMesh that the customer must receive each item fresh and hot, giving the feeling of having been recently brewed or baked.

Now that we have an example to work with, let's see how we'd implement the CoffeeMesh application as a monolith and as microservices, and discuss the merits and challenges of each approach.

1.1.4 Microservices vs monoliths

This section expands the definitions of the previous sections about microservices and monoliths by comparing the advantages and limitations of each approach. You're probably already familiar with one or more web development frameworks in Python such as Flask, Django, or Starlette. You're also probably familiar with websites which are implemented as a monolith; that is, as a single application using one of those frameworks. In this section, we'll see how such implementations differ from a microservices approach, by showing how we'd design the CoffeeMesh website as a Django application, and how we'd design it using a microservices approach. We'll use Django in this example since it's one of the most popular web development frameworks in Python. If you are less familiar with Django, see the sidebar for a quick overview of how it works.

THE DJANGO FRAMEWORK

Django is one of the most popular frameworks for web development in Python. It implements a version of the Model-View-Controller (MVC) pattern which is sometimes called Model-Template-View (MTV) or Model-View-Template (MVT). The idea behind the MVC pattern is to organize applications in three layers:

- Model: this layer provides an interface to the data. This interface can in turn be implemented in a number of different patterns, such as the Active Record Pattern, which represents database models as classes. Django implements the Active Record Pattern in its model layer.
- View: this layer provides functions or methods which can render data in a specific format, such as an HTML template.
- Controller: captures user input and maps it to specific functions or methods in the views.

In the case of Django, we say that it implements the Model-Template-View pattern because the Templates play a major role in it. Django templates are HTML stubs which are used by the views to dynamically render content.

A typical Django application is structured under a main directory called project directory. Inside the project directory, there is a main folder named after the project itself, which contains the main settings for the project. Alongside there are a number of folders representing different applications. In the context of Django, an application is a library which implements a specific component of the website, and can implemented in a generic way so that it can be reused in other projects. Figure 1.1 illustrates a possible structure for the CoffeeMesh application implemented in Django.

THE MONOLITH: ADVANTAGES AND DISADVANTAGES

In this section, we show how we would structure the CoffeeMesh website as a Django application, and discuss the benefits and challenges of this approach. Django uses the concept of project as the main namespace for your website, and within a project you can structure your code into applications. Django applications are subdirectories which contain implementations of specific components of the website. How can we break the CoffeeMesh website down into such components? Chapter 3 offers a detailed analysis of the components of the CoffeeMesh website, but for the purposes of the present discussion, we'll assume that the CoffeeMesh website consists of the following components:

- A products application to hold information and logic about the different varieties of products that the CoffeeMesh website offers, including their available stock and price.
- An orders application which encapsulates all logic necessary to take, process, and manage orders.
- A payments application to process payments.
- A kitchen application which takes care of interfacing with the automated production system of CoffeeMesh to ensure the items ordered by the customer are produced.
- A delivery application which takes care of arranging the deliveries.
- A users application to manage user accounts and authentication.

Each of these components will represent a Django application, with their own URLs, views, template, and specialized logic for their respective domains. Listing 1.1 offers an illustration of what the directory structure for this Django project looks like.

Listing 1.1 Structure of the CoffeeMesh Django project

```
coffee_mesh/
    manage.py
    coffe_mesh/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
    products/
        …
    orders/
    payments/
    kitchen/
    delivery/
    users/
```

Implementing an application as a monolith is in many cases convenient. For one, having all the code within the same codebase makes it easier to access data and logic across different applications. And because everything runs within the same process, it is easier to trace errors and bugs across the whole website: you only need to place a few breakpoints in different parts of your code, and you will get a detailed picture of what is going on when something goes wrong. Besides, because the whole codebase falls within the scope of the same project, you can leverage the productivity features of your favorite development editor when importing logic from different applications, and also to debug and profile your code.

However, as the codebase grows and becomes more complex, there comes a time when this type of architecture starts to show limitations. This happens when the codebase grows to a point where it becomes difficult to manage, and when finding your way through the code becomes arduous. Besides, one of the advantages highlighted previously, namely the ability to reuse code from other applications within the same project, often leads to the development of tight coupling among components.

Tight coupling occurs when a component depends on specific implementation details of another component. For example, imagine that the products application in the above Django project exposes a number of classes which represent the different ingredients that you can use to prepare a coffee. Imagine also that the payments application needs to consume instances of these classes in order to obtain information about the pricing of each ingredient. Each class object knows how to calculate its price depending on the amount of each ingredient. And each ingredient expresses its amount in different ways: milk in milliliters, cinnamon in milligrams, eggs in units[5], and so on.

It would be onerous for the payments application to know all of these things, and have to specify the specific unit of measure depending on the ingredient, or have to call different methods for different ingredients. That would create a tight dependency between the payments application and the specific implementation details of the classes that live under the products application. If, at some point, we want to change the units that we use to calculate the price of a specific ingredient, we will have to make changes to the payments application as well. Failing to do so will make our application behave in unexpected ways, i.e. it will introduce side-effects. A side-effect occurs whenever a code change has unintended consequences that we did not expect and have nothing to do with the intended change.

A better way is having all the ingredients classes expose the same `calculate_price()` method with an `amount` argument, and leave it to each class to interpret the amount in the corresponding unit.

Tight coupling makes it difficult to ensure that there is separation of concerns, it introduces side-effects, and eventually makes code changes more difficult and riskier. Tight coupling discourages refactoring, which means the existing codebase is never or rarely improved. The application becomes a *big ball of mud*, a term used by Brian Foote and Joseph Yoder to describe applications which are

> **"a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated." (http://www.laputan.org/mud/mud.html#BigBallOfMud)**

At the same time, every part of the monolith has to be tested, and as we add new features to it, the test suite grows unwieldy and it takes an unwarranted amount of time to complete. As a consequence, deployments become slower and encourage developers to pile up changes within the same release. This, in turn, makes releases more challenging. Because many changes are released together, if a new bug is introduced in the release, it is difficult to spot the specific change that introduced the bug and to roll it back. And because the whole application runs within the same

[5] Admittedly not the most common ingredient to make coffee, but some coffee recipes do have eggs, such as Vietnamese Egg Coffee. You can argue that you could just put all the yolks on one side and all the whites on another side, and use milliliters or milligrams to measure them. Just bear in mind that I'm not trying to teach you how to make interesting coffee recipes, but how to build microservices, so stick with me!

process, when you have to scale the resources for one component, you are scaling for the whole application (if you can scale at all). Also, if you have components that could be optimized with different hardware or server configurations, you simply cannot do that since the whole codebase ships together. Long story short, code changes become more and more difficult and deployments become a pain. How can microservices help us address these issues? The next section discusses how microservices help us overcome some of these shortcomings.

MICROSERVICES TO THE RESCUE!

In this section, we'll see how microservices help us overcome some of the challenges that come with monolithic applications. Before we delve into that discussion, I'd like to emphasize that microservices are not a panacea of benefits without limitations. Microservices bring issues of their own, and later in this chapter we'll see some of the limitations and challenges that they pose.

Microservices help us address many of issues associated with monolithic applications by enforcing a strict separation among components. When you implement an application using microservices, each microservice runs in a different processes, often in different servers or virtual machines, and can have completely different deployment models. As a matter of fact, they can be written in completely different languages (it does not mean they should!). Communication across services happens through lightweight protocols, such as web APIs, message queues, or remote procedure calls. When you need to fetch data from another service, you cannot simply import a data model from that service: you have to make an API call to the service. This creates strict boundaries among services, which means you cannot create tight coupling among their codebases. Figure 1.2 illustrates this concept by comparing the CoffeeMesh application architected as a monolithic application (left side of the illustration) and as microservices (right side of the illustration). For the sake of simplification, we have assumed that each component of the Django application that we showed earlier (refer to listing 1.1) can be mapped to an independent service in a microservices architecture.

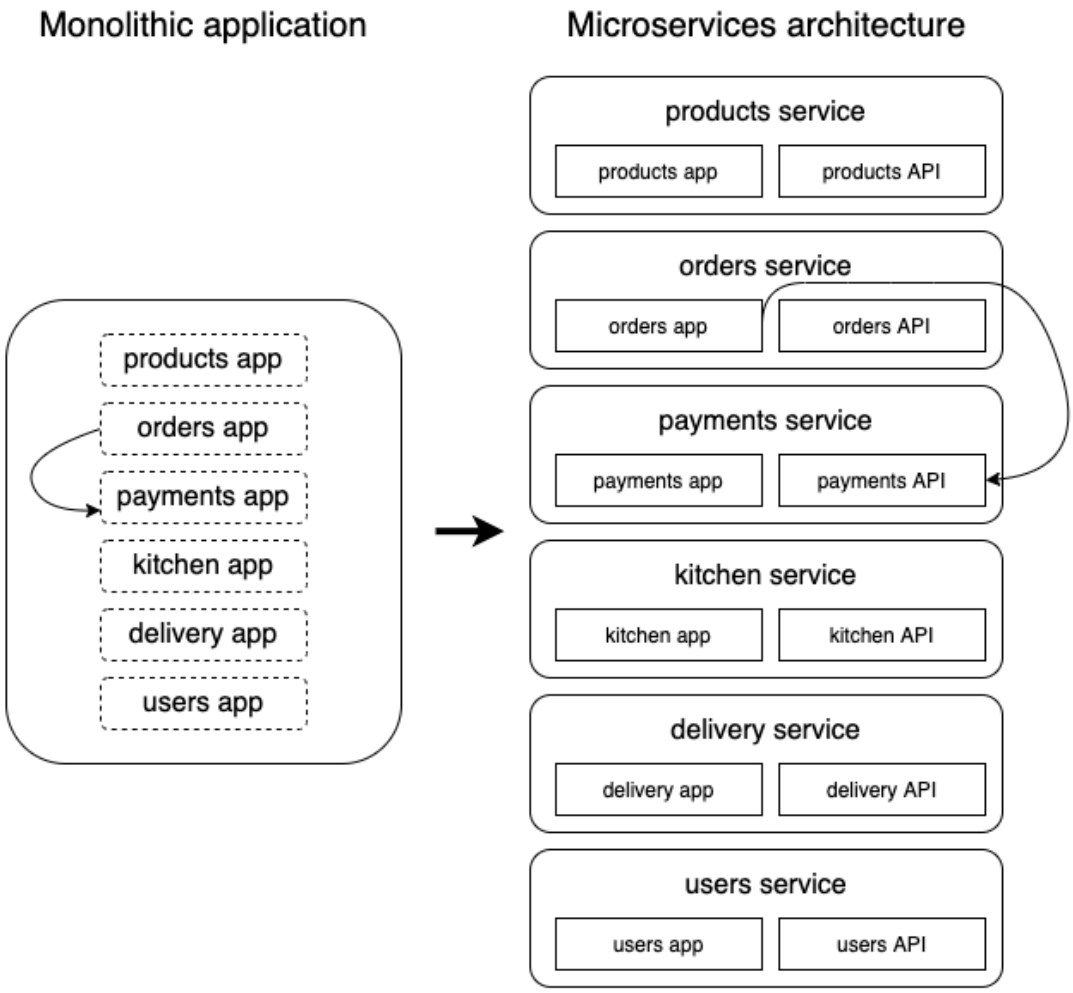


Figure 1.2. Monolith architecture (on the left) vs microservices architecture (on the right). The monolith represents the CoffeeMesh website implemented as a Django application as described in the previous section. The microservices architecture on the right shows an implementation where every component of the Django application maps directly to a service. In the monolith, we can use functionality from any component by importing functions or classes within the same codebase. In microservices, using functionality from another component requires an API call. The boundary created by the API makes it very difficult to create dependencies among components at the code level.

In Figure 1.2, the orders component is using some functionality from the payments component to process a payment. As you can see from the figure, sharing functionality across components in a monolithic application is simple, and it only requires importing a function or a class from another module. However, in a microservices scenario, in order to use functionality from another service we have to make an API call. This means that the implementation of the service whose functionality we want to use, the payments service in the case of Figure 1.2, is hidden behind the API, and therefore we cannot create tight coupling between the implementation of the orders service and

the payments service. Beware, it's still possible to couple the implementation of the orders service to the implementation of the payments API, but as long as you keep the API layer separate from the application layer, you'll always be able to make changes to the application without breaking the API and therefore the integration with other services. In the following chapters, we'll see how this is possible.

Figures 1.3 and 1.4 make this concept clearer. In Figure 1.3, the orders application from the Django project for the CoffeeMesh website imports a function `process_payment()` from the payments application, and uses it to process a payment. This rises the possibility for potential coupling between the orders application and the payments application, whereby changes to the `process_payment()` function might require changes to the orders application.

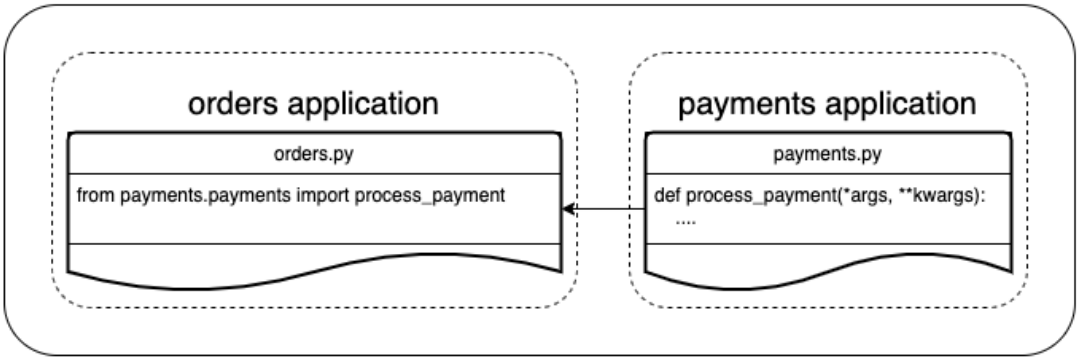


Figure 1.3. In a monolith, such as a Django application, logic can be shared across the applications by just importing functions or classes from other applications. In this illustration, the orders.py module from the orders application imports the `process_payment()` function from the payments.py module, which lives within the payments application.

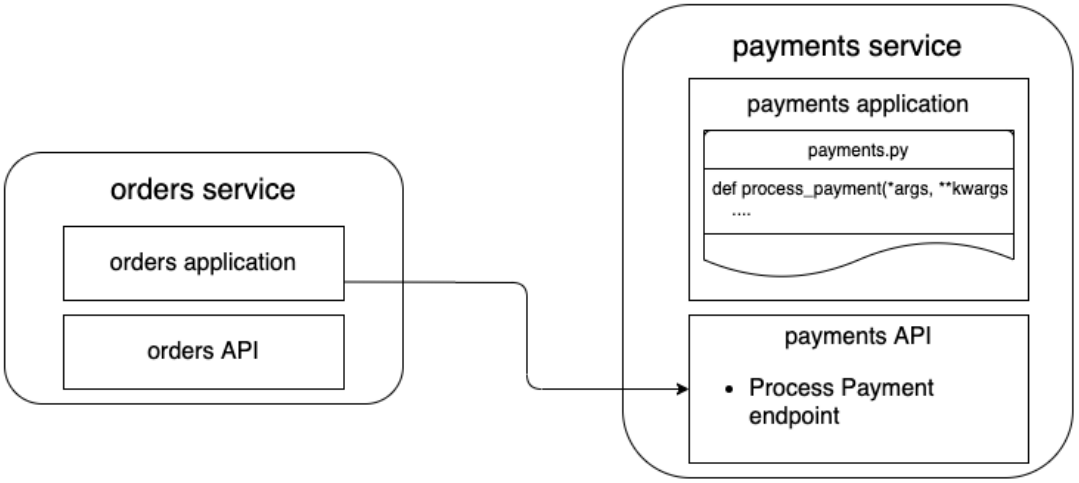


Figure 1.4. In a microservices scenario, each service runs in a different process, so it isn't possible to import code

from those services in order to share logic or data. Instead services interact with each other through API calls. In this illustration, the application layer of the payments service implements a `process_payment()` function. The orders service needs to use the capabilities implemented by that function in order to process payments, however it isn't able to import the function in order to use it, and instead has to send a request to the payments API.

In Figure 1.4, which represents a microservices scenario, this is not possible, since the orders and the payments services run in different processes and do not share the same codebase. As a matter of fact, they do not even share the same database. Instead, you have to interact with the API exposed by each of these services in order to obtain the data that you need. You cannot simply import code from the payments service into your orders service to obtain the data. Suddenly, the risk of creating tight code dependencies across applications has disappeared. You can make changes to any part of the application layer of the payments service, and as long as you keep exposing the same endpoints with the same features, the orders service will not notice any difference. Beware, this is not to say that microservices are an antidote against tight coupling – you can have tightly coupled microservices in the form of distributed monoliths, but we will see in later chapters how to avoid that.

The APIs exposed by microservices are of utmost importance. APIs represent your interfaces to the different capabilities of your microservices, and these interfaces contain a number of rules about how you can interact with each service. In order words, APIs represent the *contract* between different services. This means that any service consuming the API exposed by another service knows exactly how it needs to call that API, and what kind of responses it will get. Because APIs are your interface to the application running in a service, they are crucial for microservices integrations.

In their seminal article "Microservices", Martin Fowler and James Lewis popularized the idea that the best strategy for integrating microservices is by having them expose *smart endpoints*, and making them communicate with each other through *dumb pipes* (https://martinfowler.com/articles/microservices.html). This idea is inspired in the design principles of Unix systems. These principles, as documented by Doug McIlroy, one of the early contributors to the development of Unix, establish that:

- A system should be made up of small, independent components which do only one thing.
- The output for every component should be designed in such a way that it can easily become the input for another component.

Unix programs communicate with each other using pipelines, which is a simple mechanism for passing messages from one application to another. To illustrate this process, think of the following chain of commands, which you run from the terminal of a Unix-based machine (for example, a Mac or Linux machine):

```
$ history | less
```

The `history` command shows you the list of all commands that you have recently run using your current bash profile. The list of commands can obviously get long, so you may want to paginate its output using the `less` command. In order to pass data from one command to the other, you use the pipe character (`|`), which will instruct the shell to capture the output from the `history` command and pipe it to the input of the `less` command. We say that this type of pipe is "dumb" because its only job is passing messages from one process to another. As you can see in figure 1.5, in the case of web APIs data is exchanged through an underlying data transfer/transport layer

(HTTP over TCP/UDP in most cases). This data transport layer knows nothing about the specific API protocol that we are using, and therefore it represents our "dumb pipe", while the API itself contains all the necessary logic to process the data.

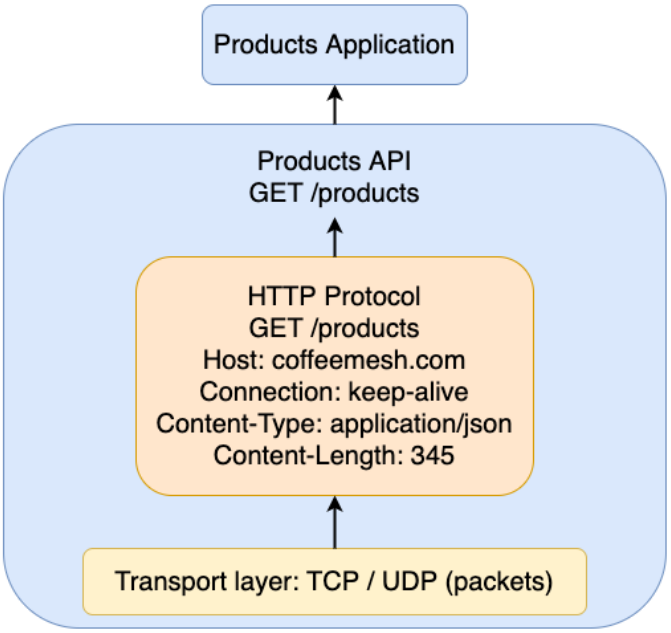


Figure 1.5. An API communicates over a data transfer protocol or transport layer. In this illustration, the Products API is accepting requests over HTTP, leveraging HTTP specific features such as HTTP verbs (in this case, GET). HTTP in turn is using TCP/UDP as a data transport layer.

APIs must be stable, and behind them you can change the internal implementations of any service as long as they keep honoring their contracts. This means that the consumer of an API must be able to continue making calls to the API in the exact same way as before, and it must continue getting the exact same responses. This leads to another important concept in microservices architecture, which is *replaceability*[6]. The idea is that you should be able to completely replace the codebase that lays behind an endpoint, yet the endpoint, and therefore communication across services, will still work.

Because microservices contain smaller codebases than a monolith, and because their logic is self-contained and defined within the scope of a specific business capability, it is easier to test them, and their test suites run faster. Because they do not have dependencies with other components of the platform at the code level (except perhaps for some shared libraries), their codebases are clearer, and it is easier to refactor them. This means the codebase can get better over time, which makes it more maintainable. All of this means we can make small changes to the codebase, and release on every Pull Request merge. This means releases are smaller, which means

[6] Sam Newman, *Building Microservices* (Sebastopol CA, O'Reilly, 2015), pp. 7-8.

they are more controllable, and if we spot a bug, they are easier to roll back. These features lead to a situation where you can make several releases per day, or even per hour.

1.1.5 Microservices today and how we got there

In many ways, microservices are not something new[7]. Many companies have been implementing and deploying components as independent applications well before the concept of microservices became popular. They just did not call it microservices. Werner Vogels, CTO of Amazon, explains how Amazon started to experiment with this type of architecture in the early 2000s. By that time, the codebase for the Amazon website had grown into a complex system without a clear architectural pattern, where making new releases and scaling the system had become serious pain points. So they decided to look for independent pieces of logic within the code, and to single them out into independently deployable components, with an API in front of them. As part of this process, they also identified the data that belongs to those components, and made sure that other parts of the system could not access such data except through an API. They called this new type of architecture "service-oriented architecture"[8]. Netflix also pioneered this type of architectural style at scale, and they referred to it as "fine-grained Service Oriented Architecture"[9].

The term microservice grew in popularity in the early 2010s to describe this type of architecture. For example, James Lewis used this concept in a presentation at the 33rd Degree conference in Krakow in 2012, under the title "Micro-services – Java, the Unix way"[10]. In 2014 the concept was consolidated with a paper written by Martin Fowler and James Lewis about the architectural features of microservices, as well as the publication of Sam Newman's seminal book *Building Microservices: Designing Fine-Grained Systems* (O'Reilly Media, 2015).

Service-Oriented Architecture (SOA)

Service-oriented Architecture (SOA) is a software architectural pattern which emphasizes the organization of an application into independent, self-contained components which may communicate with each other a communication protocol of choice, such as HTTP, over a network. At a high level, SOA and microservices architecture might seem to be related, or at least microservices architecture seems like a natural evolution from SOA. When it comes to implementation details, however, SOA doesn't necessarily emphasize the same features that characterize microservices. For example, where microservices emphasize the use of dump pipes for communication, SOA uses smart pipes, such as Enterprise Service Bus (ESB). Where microservices emphasize the use of one database per service and discourages sharing data across services, SOA has a global and shared data model.

Today, microservices are a widely used architectural style. Most companies where technology plays an important role are already using microservices or moving towards its adoption. It is also common for startups to start implementing their platform using a microservices approach. However, microservices are not for everyone, and although they bring substantial benefits, as we have shown above, they also carry considerable challenges, as we shall see in section 1.5 of this chapter.

[7] For a more comprehensive analysis of the history of microservices architecture and its precursors, see Dragoni, Nicola; *et al.*, "Microservices: yesterday, today, and tomorrow", *Present and Ulterior Software Engineering*, pp. 195-216,arXiv:1606.04036.
[8] Werner Vogels, "Amazon and the Lean Cloud", HackFwd Build 0.7, Berlin, September 2011 (https://vimeo.com/29719577).
[9] Allen Wang and Sudhir Tonse, "Announcing Ribbon: Tying the Netflix Mid-Tier Services Together", Netflix Technology Blog, Jan 18 2013 (https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62).
[10] James Lewis, "Micro-services – Java, the UNIX way", 33rd Degree Conference, Krakow, 2012 (available at https://vimeo.com/74452550).

1.2 What are web APIs?

In this section we will explain what web APIs are. You will learn that a web API is a specific instance of the more general concept of Application Programming Interface (API). It is important to understand that an API is just a layer on top of an application, and that there are many different types of interfaces. For this reason, we will begin this section defining what exactly an API is, and then we will move on to explaining the defining features of a well implemented web API.

1.2.1 What is an API?

We will begin this section by defining what an API is and how it is different from the application layer. We will use the example of the well-known CLI tool `cURL` to illustrate the concepts.

API stands for Application Programming Interface, and it defines the way we can programmatically interact with an application. It represents a contract. This means that the API tells you what you can do to interact with an application, and what you can expect from the application by executing those actions. An API is a layer on top of an application, and it must not interfere with the application logic. There are multiple types of application interfaces, such as command line interfaces (CLI), desktop UI interfaces, web UI interfaces, or web API interfaces. An application can have one or more of these interfaces, and therefore its logic should not be bound to a specific type of interface. In other words, you must avoid creating tight coupling between the application logic and the API layer (see Figure 1.6).

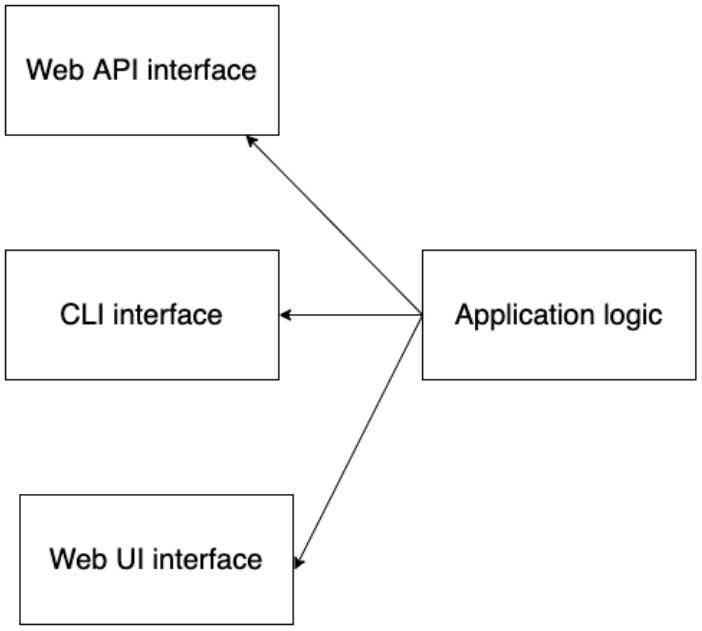


Figure 1.6. An application can have multiple interfaces, and therefore the application logic should be independent from the interface logic, and there should be no strong dependencies or tight coupling between the application layer and the interfaces layer. In this illustration, we show an application with three different types of interfaces: a web API interface; a command line interface; and a web UI interface.

To illustrate this idea, think, for example, of the popular Client URL CLI (`cURL`). `cURL` is an command line interface to an application (`libcurl`). `libcurl` implements the capabilities that we need to be able to interact with URLs, while `cURL` exposes a number those capabilities for us to use through a number of options that we can specify to trigger certain actions in the application. For example, we can use the `-L` option to instruct `cURL` to follow a redirect:

```
$ curl -L http://www.google.com
```

Or we can use the `-O` option in order to download the contents of a URL endpoint to a file:

```
$ curl -O http://www.gnu.org/software/gettext/manual/gettext.html
```

Adding the `-C` option also allows us to resume downloading content which, for some reason, was stopped. Similarly, we can use the `-X, --request`[11] option to specify a HTTP method, the `-H` option to add request headers, and the `-d` option to attach a payload in the request:

```
$ curl -X POST -d '{"title": "foo", "body": "bar", "userId": 1}' -H 'Content-Type:
       application/json' 'https://jsonplaceholder.typicode.com/posts'
```

In all of these cases, we are using the API exposed by the `libcurl` application's CLI (`cURL`). The API is well specified (to see it, run `curl --help`), and from it we know what to expect from the application when we specify different options. The `libcurl` application sits behind this interface, and nothing prevents us from accessing it directly through the source code[12], and even building additional types of interfaces for this application.

1.2.2 What is a web API?

Now that we understand what an API is, we will explain the defining features of a web API. We will explain how a web API layer works, what protocols and frameworks are available for its implementation, and how they fit within the architecture of a web application.

So, what exactly is a web API? Simply put, a web API is a web interface to an application. Web applications typically return an HTML document that can be rendered in the browser. However, that type of interface is not accessible programmatically, i.e. it does not expose commands that allow you to interact with it programmatically. With a web API, you expose an interface to the web application running in your server, that can be used to interact with it programmatically.

Web APIs are implemented using a protocol or framework of choice. There are different options available for that, such as SOAP, REST, GraphQL, gRCP, Twirp, and others that we will discuss in more detail in chapter 2. Most of these API protocols run or can run on top of Hypertext Transfer Protocol (HTTP), which is the communication protocol that underpins the Internet. HTTP is designed to be able to transfer different kinds of media type over the Internet, such as text, images, video, JSON, etc. It uses the concept of Unique Resource Locator (aka URL) to find resources in the Internet, and provides utilities that can be leveraged by API technologies to enhance the interaction with the server, such as request methods (e.g. GET, POST, PUT). We will delve into these topics in more detail in later chapters.

[11] In command line applications, it is usual to offer a shorter and a longer version of the same option, separated by a comma. The shorter version is always preceded by one dash, and the longer version is preceded by two dashes. In this example, the notation -X, --request means that this option can be used as -X or as --request.

[12] If you are curious, you can pull it from Github: https://github.com/curl/curl.

The growth of web APIs has transformed the way we build applications. For one, the wide availability of APIs in the Internet means that, when building your own products, you do not have to implement all features or functionality by yourself. In many cases, you can just use one of the existing APIs offered by different providers or vendors, and integrate it with your application. This could be the case for authentication management, artificial intelligence applications such as voice recognition or natural language processing, to deliver emails, or to obtain the coordinates of a specific address. For this and many other use cases, there is a wide availability of offer in the Internet that you can choose from.

It also means that, when building a web application, you have several choices of architecture and design. You can choose to build a traditional website with dynamically rendered views from the server, such as you would normally do with frameworks like Django. You can choose to build your application with a client-server architecture, whereby the client, typically an SPA (single page application) which interacts with the backend through an API. As part of the latter design, you can decide to open part of the backend API for public consumption (probably for a price), or you may decide to skip the frontend altogether and offer your service exclusively via web APIs.

1.3 Why does it make sense to deploy web APIs as microservices?

Now that we understand what microservices architecture is, and what the defining features of web APIs are, in this section we explain why it makes sense to deploy web APIs as microservices. To illustrate this idea, we resume the example of the CoffeeMesh application we presented earlier in this chapter, and we discuss how we can break it down into microservices with their own APIs. Along the way, we will explain the benefits that we gain from migrating the application to a microservices architecture.

You can deploy an API as part of a monolithic application, and for many use cases that can be perfectly fine. However, an API usually contains a collection of different endpoints, and often those endpoints represent different services or capabilities of the website. In that respect, it makes sense to break down the API into different microservices and deploy them separately.

Going back to our previous example, the CoffeeMesh Django project is already organized around applications which own different endpoints of the website. For example, the payments application is served under `/payments`, while the users application is served under `/users`, and the orders application is served under /orders. Each application encapsulates a specific business domain of the website, and owns specific segments of its data. Therefore, we could easily enhance or repurpose them to expose well defined API endpoints. We could then deploy all these applications together as part of a single monolith, and we would benefit from being able to share code and data across applications. However, as we saw earlier, this approach eventually leads to scenarios where it becomes difficult to manage the codebase, make changes to it, release new features, and scale the system.

Alternatively, we can break the applications apart and deploy them as independent services and avoid many of those problems. To be able to deploy each Django application as an independent service, we first need to remove the code dependencies among the applications. This is easier said than done, but for the purposes of the present discussion let's assume we are able to do that and move forward. The process of breaking an application down into smaller components is called *service decomposition*. We'll talk a lot more about service decomposition in Chapter 3, where we'll

explain two methods that we can use to decompose a system into microservices[13]. If we migrate our CoffeeMesh Django project to a microservices architecture, the result might look like something like this (see Figure 1.7 for an illustration of this migration):

- A products service which encapsulates logic and data about the products on offer and their stock availability. It exposes the /products URL path.
- An orders service which encapsulates all logic necessary to place and manage orders. It exposes the /orders URL path.
- A payments service to process payments. It exposes the /payments URL path.
- A kitchen service which knows how to interface with the automated production system. It exposes the /kitchen URL path.
- A delivery service which takes care of arranging deliveries. It exposes a /delivery URL path.
- A users service to manage user accounts and authentication. It exposes a /user URL path.

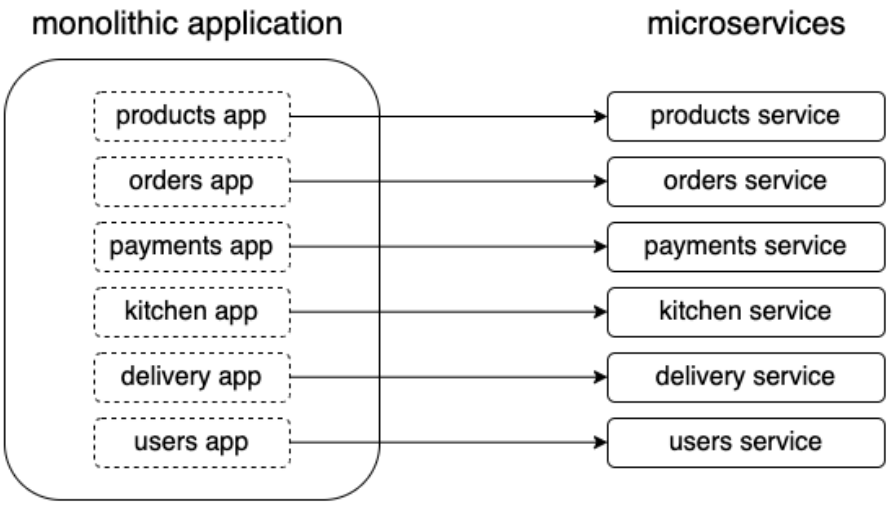


Figure 1.7. In a monolithic application, all components ship together within the same codebase, while in a microservices architecture, components are deployed separately as independent applications and run in different processes.

As you can see, this layout mirrors the structure of the Django application we presented at the beginning of this chapter. This is not to say that, when migrating a traditional Django application to a microservices architecture, you can just simply take the applications in your Django project and deploy them as microservices. You have to consider whether each application owns a specific and well-defined business domain. It may be the case that some applications must be broken down into smaller microservices components. On the other hand, it may make more sense to merge some applications within the same service.

In the microservices scenario, communication and data sharing across services takes place through APIs (please refer back to figures 1.2 and 1.4 for an illustration of this idea). For example,

[13] Please bear in mind that the approach shown in this section is simplified. Simply deploying the applications of a Django project as independent services is not a robust approach for service decomposition. For effective service decomposition strategies please refer to Chapter 3. The approach we follow in this section is an intended simplification to allow us to discuss the advantages and disadvantages of deploying web APIs as microservices.

to process a payment, the orders service has to interact with the service payments API. At this point, we are not specifying any details about how exactly these API calls should work. We are doing this intentionally, because such level of detail requires us to choose a specific technology for the implementation of the API, which is not the intention at this point. We are now just considering the pros and cons of microservices vs monoliths, and we should keep our considerations at a high level at this point.

In the microservices scenario, there are strict boundaries between the services. Each service owns a specific set of data, and other services cannot access such data except through the interface exposed by other services. This helps to prevent us from creating unnecessary coupling among the services at the level of the code. We are also able to scale each service independently, and to optimize the environment in which it runs. For example, it may be the case that the payments service performs tasks which are a lot more CPU intensive than the delivery service, which in turn may require more memory in order to load all the delivery details about each order, while the products service may require more optimized networking hardware in order to process quickly its numerous database queries and respond timely to each service which requires its data. It may also be the case that the kitchen service may need more resources during certain times of the day, while the rest of the time it can be idle. Thanks to the fact that each service runs in a different process and in a different environment, we are able to provision each of them with the most optimized configuration for their operations, and we are able to scale them (up and down) independently.

1.4 How Python makes it easier to develop microservice web APIs

In this section we will explain why it makes sense to implement API-driven microservices using Python. We will see that Python provides a rich ecosystem for the development of web APIs and for building microservices. We will use some of the tools discussed in this section over the course of this book, but we will not be able to provide examples of use for all of them. However, it is important that you are aware of the wide array of resources that we can choose from when implementing API-driven microservices in Python, and this section will give you a quick overview of the most relevant resources at the time of this writing.

Python is one of the most popular languages for developing web backend services. The Stack Overflow Developer Survey of 2019 ranks Django and Flask as two of the most popular web development frameworks (https://insights.stackoverflow.com/survey/2019). In fact, in terms of backend web development frameworks (that is, excluding frontend development frameworks such as Angular or React), Python is the only language represented more than once in the top 12 of the "Web Frameworks" ranking[14].

Python offers a rich ecosystem for web applications development, and in particular for APIs. If you are developing a Django website, you can use the Django REST framework, an excellent, battle-tested and very well documented framework for RESTful APIs (https://github.com/encode/django-rest-framework). Or you can use graphene-django if you intend to build a GraphQL API (https://github.com/graphql-python/graphene-django). If instead you are using Flask, you can choose between Flask-RESTful (https://github.com/flask-restful/flask-restful), Flask-RESTX (https://github.com/python-restx/flask-restx) or flask-smorest (https://github.com/marshmallow-code/flask-smorest) to build a RESTful API, or you can plug in

[14] Combining Django and Flask, Python appears to be the second most popular choice for web backend development.

graphene (https://github.com/graphql-python/graphene) directly into your Flask web application in order to enable GraphQL endpoints, or use Flask-GraphQL (https://github.com/graphql-python/flask-graphql) in order to facilitate the implementation. You can further use a library like marshmallow (https://github.com/marshmallow-code/marshmallow) in order to optimize your object serialization tasks. You can also use fastapi (https://github.com/tiangolo/fastapi) to build highly optimized APIs in Python, and to implement both REST and GraphQL endpoints.

The previous paragraph describes only some of the most popular choices available for web development with Python. There are many other popular libraries, such as Tornado (https://github.com/tornadoweb/tornado), Pyramid (https://github.com/Pylons/pyramid), Falcon (https://github.com/falconry/falcon), and others that we cannot cover in this book. Also, the ecosystem is growing by the day, with new choices becoming available that you should watch out for and consider. The point is that Python offers a very rich ecosystem for web application development, and therefore makes an excellent choice for the implementation of web APIs.

At the same time, Python offers a great ecosystem for building microservices. As we will see in chapter 14, when it comes to microservices, there are multiple solutions you can use for deployments, including Docker, serverless, and managed hosting services such as Heroku or AWS Beanstalk, to name a few. Python enjoys good support in most of these platforms, and in many cases, the ecosystem provides custom tooling and possibilities for optimization.

For example, if you are going to deploy with Docker, you will be able to choose between minimal Linux distributions such as Alpine Linux, and purpose-built base images for Python. You can also consider deploying on AWS Lambda, the serverless computing platform provided by AWS, and you will be pleased to discover that, at the time of this writing, it currently supports Python versions 3.6, 3.7, and 3.8. If this is not sufficient for your needs, you can use Lambda Layers together with the Lambda Runtime API, which are additional services that AWS provides to help you to create support for additional runtimes, such as other versions of Python. When it comes to serverless deployments in AWS, you are also able to use tools such as Chalice (https://github.com/aws/chalice) and Zappa (https://github.com/Miserlou/Zappa), which are implemented in Python, thereby giving you the possibility to enhance their implementations and customize your builds. You can also deploy serverless applications written in Python to Azure using Azure Functions, and to Google Cloud using GCP Functions. If you're considering deploying to Heroku[15], you will find that it has support for all the major Python frameworks, including Django, Flask, Pyramid, and so on.

The bottom line is that, when it comes to building and deploying microservice APIs, Python gives you a rich ecosystem, with a wide variety of frameworks that can help you boost your development productivity, as well as a great variety of tools and services to customize and optimize your deployments.

1.5 Microservices-specific challenges

As we have seen earlier, Microservices come with substantial benefits of their own. However, they also come with significant challenges at many different levels. In this section, we discuss some of the most important challenges that microservices bring, which we will classify into six main categories:

[15] Heroku is a cloud platform as a service (PaaS) provider. It allows you to upload your applications, and with minimal configuration get them running, while they take care of managing and provisioning computing resources. You can learn more about Heroku from their website: https://www.heroku.com/.

- Effective service decomposition
- Microservices integration tests
- Handling service unavailability
- Distributed transactions
- Increased operational complexity
- Infrastructure overhead

All the problems and difficulties that we discuss in this section can be addressed with specific patterns and strategies that we detail in later chapters. The idea here is to make you aware that microservices, by themselves and in themselves, are not magical cure against all of the problems that monolithic applications present.

1.5.1 Effective service decomposition

One of the most important challenges when designing microservices is ensuring a clean breakdown of your platform into loosely coupled yet sufficiently independent components with clearly defined boundaries. You can tell whether you have unreasonable coupling between your services if you find yourself changing one service whenever you change another service. In such situations, either the contract definition between services is not resilient, or there exist enough dependencies among both components to justify merging them together. Failing to break down a system into independent microservices can result in what Chris Richardson, author of *Microservices Patterns* (Manning, 2018), calls a "distributed monolith," a situation where you combine all the problems of monolithic architectures with all the problems of microservices, without enjoying the benefits of any of them.

1.5.2 Microservices integration tests

We said before that microservices are usually easier to test, and that their test suites generally run faster. Microservices integration tests, however, can become significantly more difficult to run, especially in cases where a single request involves collaboration among several microservices. When your whole application runs within the same process, it is fairly easy to test the integration between different components, and most of it will simply require well-written unit tests. In a microservices context, in order to test the integration among multiple services, you need to be able to spin up all of them with a setup similar to the production environment.

You can use different strategies to test microservices integrations. The first step is making sure that each service has a well-documented and correctly implemented API. You can accomplish this by writing unit tests against each of the endpoints that each service exposes, using the API documentation for validation. You must also ensure that the API client is consuming the API exactly as dictated by the API documentation. You can write unit tests for the API client using the API documentation to generate mocked responses from the service. Finally, none of the above tests will be sufficient without a full-blown end-to-end test that runs the actual microservices making calls to each other.

1.5.3 Handling service unavailability

We have to make sure that our applications are resilient in the face of service unavailability, connections and request timeouts, erroring requests, and so on. For example, when a customer places an order through the CoffeeMesh website, a chain of requests between services unfolds to

process and deliver the order, and any of those requests can fail at any point. Let's inspect the chain of requests that takes place when a user places an order (see Figure 1.8 for an illustration of the chain of requests):

1. User places an order and pays for it. The order is placed using the orders service API, and to process the payment, the orders service makes a request to the payments service.
2. If payment is successful, the orders service makes a request to the kitchen service to schedule the order placed by the customer.
3. Once order has been produced, the kitchen service makes a request to the delivery service to schedule the delivery.

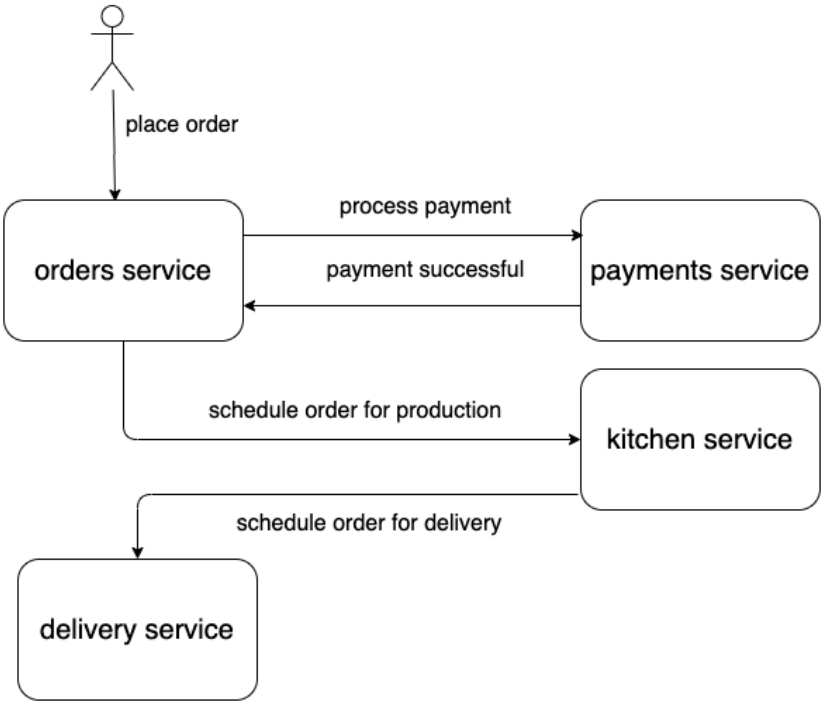


Figure 1.8. Microservices must be resilient to events such as service unavailability or requests timeouts. This figure illustrates the complex chain of requests triggered when a customer places an order: the orders service makes a request to the payments service to process the payment, and if payment is successful, the orders service schedules the order for production by making a request to the kitchen service. Once the order has been produced, the kitchen service makes a request to the delivery service to schedule the delivery.

In this complex chain of requests, if one of the services involved fails to respond as expected, this can trigger a cascading error through your microservices that can leave the order unprocessed or in an inconsistent state. For this reason, it is important to design microservices in such a way that they can deal reliably with failing endpoints. And again, a clean and clear separation of concerns among services, together with solid contracts, should minimize the risk of breaking changes

affecting other services. Our end to end tests should consider these scenarios and test the behavior of our services in those situations.

1.5.4 Distributed transactions

Collaborating services sometimes have to handle distributed transactions. Distributed transactions are transactions which require the collaboration of more than one service in order to update the state of one or more entities. In the CoffeeMesh application, the products service manages the products on offer and their inventory. The inventory of a product has to be updated whenever a user places an order, so that other users can get an updated list of available choices when they visit the website. Specifically, we want to update the inventory once the payment has been successfully processed. The successful processing of a payment therefore involves the following actions (see Figure 1.6 for an illustration of this process):

1. Update the status of the order to `active`.
2. Interface with the kitchen service to schedule the order for production.
3. Update the inventory for the purchased products to reflect their current availability.

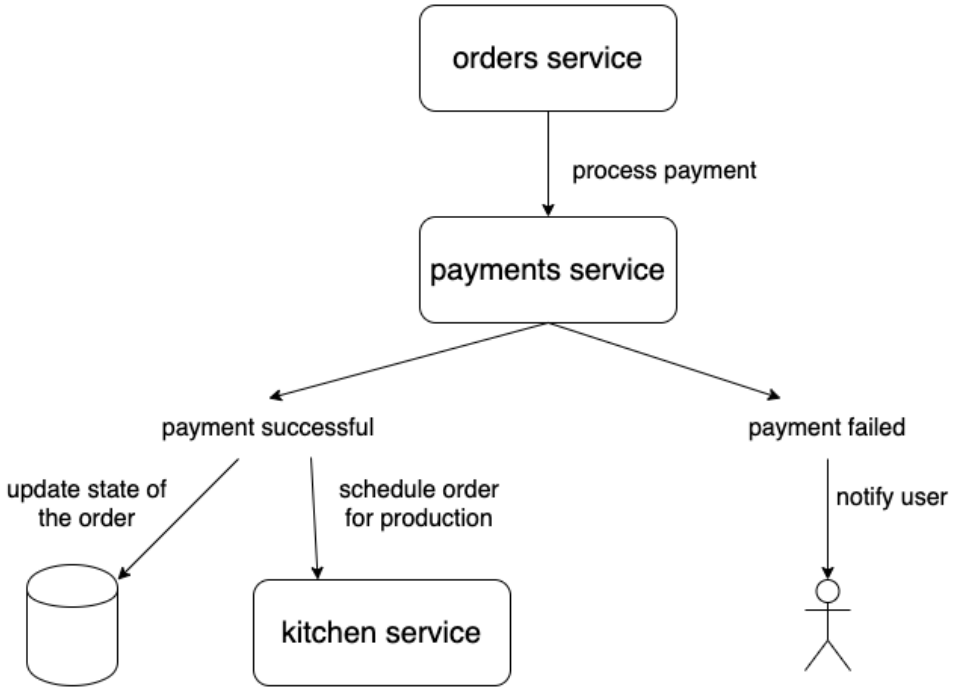


Figure 1.9. to return a successful response to the user, the orders service must successfully communicate with the payments service, the kitchen service, and the products service. If any of these interactions with other services fails, the order service must be able to handle the failure and provide a meaningful response to the user.

All of these operations are related, and must be orchestrated such that either they succeed or fail all together. We can't have an order successfully paid without correctly updating its status, and we shouldn't schedule its production if payment fails. We may want to update the availability of the ingredients at the time of making the order, and if payment fails later on, we want to make sure we rollback the update. If all these actions happened within the same process, managing the flow would be fairly straightforward, and Python gives us specific control flow utilities that makes it even easier, such as context managers.

In the context of microservices, the challenge is ensuring that we have a robust communication process among services, such that we know exactly what kind of error happens when it does, and take appropriate measures in response to it. In the case we described above, you may want to make asynchronous requests to the payments service while asynchronously updating the status of the order and the inventory. Eventually, if the payment isn't successful, we can roll back the update the status for the order and the inventory.

Of course, the latency of inter-service requests means that the system will never be in a strongly consistent state. In the world of microservices, you have to embrace eventual consistency. Eventual consistency means that, as long as a resource is being updated, there is no guarantee that access to the resource will give you the latest state of it. However, if updates to the resource stop for a while, all readings of will eventually return the latest state. Eventual consistency is fundamental in distributed computing, and the design of your applications must account for it and deal with it.

1.5.5 Increased operational complexity

Another important challenge that comes with microservices is the increased operational complexity they add to your platform. When the whole backend of your website, or most of it, runs within a single Django application, there is only one process that you have to monitor and trace. When you have a dozen microservices, you have to monitor each service, collect and analyze their logs, create alerts for each of them and ensure they have scaling and failover mechanisms adjusted to their needs. In the case of services that work collaboratively to serve certain requests, you also have to be able to trace the cycle of the request as it goes across the different services in order to be able to detect errors during the transaction. In chapter 14, we will discuss these challenges at greater length and explore some of the solutions that you can take to tackle them.

1.5.6 Infrastructure overhead

Finally, microservices come with an overhead in terms of infrastructure. The infrastructure for each service has to be developed, configured, deployed, and managed. And this includes not only the provisioning of servers to deploy the services. It includes also the log aggregation streams, the monitoring systems, the alerts, the self-recovery mechanisms, and so on and so forth. And because every service owns its own database, they also require database setups with all the features needed to operate at scale. Also, there are cases when a new deployment requires infrastructure changes, which have to be promptly applied in order for things to work as expected. And it is not unusual that a new deployment changes the endpoint for a microservice, whether it's the IP, its base URL, or its specific path within a generic URL. This means that we need to use a service discovery mechanism that all services can rely upon.

When Amazon first started their journey towards a microservices architecture, they discovered that development teams would spend about 70% of their time managing infrastructure. This is a

very real risk that you face if you do not adopt best practices for infrastructure automation from the beginning. And even if you do, you are likely to spend a significant amount of time developing custom tooling to manage your services effectively and efficiently. We will not cover the development of such tooling in this book as it belongs more to the domain of DevOps, but you can learn more about it from Jamie Riedesel's *Software Telemetry* (Manning, 2021).

1.6 What you will learn in this book

To make the most out of this book, you should be familiar with Python and a Python development framework, such as Django or Flask. You do not need to have knowledge of web APIs or microservices, as we will explain these technologies in depth. It would be useful if you are familiar with the model-view-controller (MVC) pattern for web development, or its Django version, known as the model-template-view (MTV) pattern, as we will draw comparisons with these patterns from time to time in order to illustrate certain concepts and ideas. Basic familiarity with AWS will be useful to get through the chapters dealing with operational topics, but again we will do our best to explain every concept in detail.

This book shows you how to develop API driven microservices with Python through a hands-on approach. You will learn

- strategies for defining the application boundaries of a microservice, and for making sure you don't create unwarranted dependencies between services
- useful design patterns to help you tackle the problems that come with the adoption of microservices architecture
- how to implement APIs in Python using a documentation-driven approach. We provide examples with the two of the most popular protocols for the implementation of APIs: REST and GraphQL
- how to define the domain of a service by applying the design principles for context boundary from microservices
- how to create a meaningful API to expose the capabilities of our services, and how to define the API formally in a standard specification format, such as Swagger/OpenAPI for REST and Schema Definition Language for GraphQL
- how to use documentation-driven development in order to leverage the API documentation in the implementation stage, to automatically load validations for incoming requests and for outgoing responses. API specifications will come in handy as well when designing tests for our services.
- how to protect your API using real-world solutions for user authentication and request validation. We will look into some of the main security problems that APIs face, and how to reduce our exposure to them.
- how to deploy your microservices using serverless deployments in AWS.
- how collect logs from your microservices, how to monitor them, and how to trace distributed requests across collaborating microservices.

By the end of this book, you will be familiar with the benefits that microservices architectures bring for web applications, and also the challenges and difficulties that come with them. You will know how to implement APIs following a documentation-driven approach, and you will be prepared to define the domain of an API with clear application boundaries. Although the journey towards API

driven microservices can hardly be accomplished with just one book, hopefully you will be well on your way by the end of this one.

1.7 Summary

- Microservices are an architectural style that emphasizes the idea of developing each component of a system as an independently deployable application, which allows us to develop and deploy each service faster and more reliably
- In a monolithic architecture, the whole application logic stays within the same codebase, and runs within the same process, while in microservices each component runs in a different process. Because each microservice runs in a different server and/or environment, we optimize the environment for the application, and scale according to its needs
- Each microservice defines a well specified domain or business capability and has clear application boundaries. In particular, it does not have dependencies on other microservices, which means microservices are loosely coupled and therefore can grow, evolve and change without creating breaking changes in other components
- Microservices talk to each other using smart endpoints and dumb pipes, which means microservices don't rely on a single point of failure to distribute messages across the system: every microservice is responsible for the messages it sends and receives
- An API is an interface to an application, and it specifies how we can interact with the application and what we can expect from it: it represents a contract, which means that API consumers can rest assured that they will always obtain the expected responses, regardless of internal changes in the service
- A web API is an interface to an application that runs in a web server, it's different from the application layer and its implementation must be independent from it
- Web APIs can be implemented using a wide variety of technologies and protocols, including REST, GraphQL, and gRCP: there is great variety of choice for all cases and scenarios
- APIs are typically organized around different endpoints of a website, and each endpoint typically represents an independent business capability. Therefore, it makes sense to deploy APIs as microservices
- Python provides a rich ecosystem for the development of web APIs and microservices, and therefore makes an excellent choice for the implementation of API driven microservices
- Microservices are no silver bullet against the issues facing modern websites, and in fact come with problems of their own; you have to apply the right patterns to avoid ending up with a distributed monolith
- Distributed transactions across multiple microservices pose a challenge when trying to ensure the consistency of our data. To deal with this, you must embrace eventual consistency
- Tracing distributed transactions across multiple microservices is challenging but of utmost importance in order to gain visibility of underlying bugs and errors
- Microservices come with an additional infrastructure overhead, so before adopting microservices, you have to make sure you are ready to take on such overhead

2

A basic API implementation

This chapter covers

- Reading and understanding the requirements of an API specification
- Structuring our application into a data layer, an application layer, and an interface layer
- Implementing API endpoints using FastAPI
- Implementing data validation models (schemas) using Pydantic

In this chapter, we implement the API for the orders service, which is one of the microservices of the CoffeeMesh website, the project that we introduced in chapter 1. CoffeeMesh is an application that makes and delivers coffee on demand at any time, wherever you are. In this chapter, we'll implement the API for one of the services in the CoffeeMesh platform: the orders service, which allows customers to place orders. In doing so, you will get an early peek into the concepts and processes that we dissect in more detail throughout this book. In this chapter, I go quickly through the steps and concepts involved in the implementation of a microservice, and build an intuition of the main ideas that we will cover in more detail in later chapters.

The code for this chapter is available under the folder `ch02` of the GitHub repository provided with this book.

2.1 Introducing the API specification

This section discusses the requirements for implementing the API of the orders microservice. The orders microservice allows us to manage orders made through the CoffeeMesh website. It's a core service of CoffeeMesh and it integrates with other services of the platform, as shown in figure 2.1. You'll find an explanation of how we broke down the CoffeeMesh platform into microservices in chapter 3.

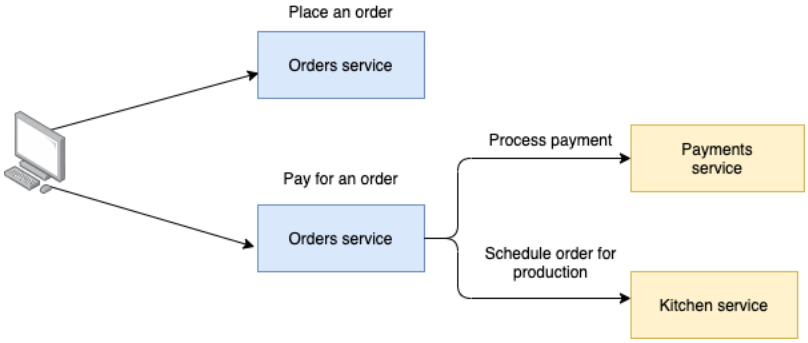


Figure 2.1 Through the orders service, a user can perform several actions, such as placing an order and paying for it. To accomplish these tasks, the orders service interacts with other microservices of the CoffeeMesh platform which are specialized in the processing of those actions.

Through the API exposed by the orders service, we can place orders, update them, retrieve their details, or cancel them. The API specification for the orders API is available in appendix A of this book and also under the folder `ch02` of the GitHub repository provided with this book. A thorough explanation of how such specification is developed is offered in chapter 4. The API specification describes a REST API with four main URL paths, where each of the paths contains endpoints which expose specific capabilities of the orders service (see figure 2.2 for additional clarification):

- `/orders`: allows us to retrieve lists (GET) of orders and to create orders (POST).
- `/orders/{order_id}`: allows us to retrieve the details of a specific order (GET), to update and order (PUT), and to delete an order (DELETE).
- `/orders/{order_id}/cancel`: allows us to cancel an order (POST).
- `/orders/{order_id}/pay`: allows us to pay for an order (POST).

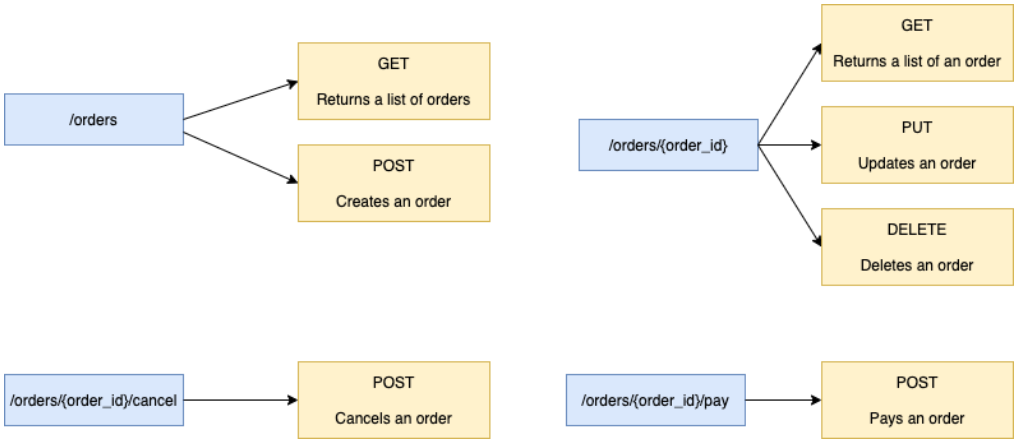


Figure 2.2 The orders API consists of four URL paths, each of which implements endpoints with specific capabilities. Each endpoint is represented by the combination of URL path plus HTTP verb. For example, the `/orders` URL path exposes 2 endpoints: a GET endpoint, which returns a list of orders, and a POST endpoint, which can be used to place an order.

The API specification for the orders API contains a detailed specification of the payloads, parameters, and expected responses for each endpoint. Of particular importance is the collection of `schemas` defined within the `components` section of this document. These schemas contain all the information we need to implement the validation mechanisms for incoming payloads sent by a consumer of the API. Such validation is fundamental to ensuring that orders are created and updated with the right attributes and data types in the server. For example, the schema for `OrderItemSchema`, which you can find reproduced in listing 2.1, specifies that the `product` and the `size` properties are required, but the `quantity` property is optional. When the `quantity` property is missing from the payload, the default value is 1. Our API implementation must therefore validate the presence of the `product` and the `size` properties in the payload before we try to create the order. With this information under the hood, let's now give a brief overview of the patterns that we'll use to structure our application.

Listing 2.1 Specification for OrderItemSchema

```
OrderItemSchema:
  type: object
  required:
    - product
    - size
  properties:
    product:
      type: string
    size:
      type: string
      enum:
        - small
```

```
        - medium
        - big
    quantity:
      type: integer
      default: 1
      minimum: 1
```

2.2 High-level architecture of the orders application

Now that we have discussed the requirements for implementing the orders application, I would like to take a moment to explain the approach that we will take to structuring the code.

> **SEPARATE THE API LAYER FROM THE BUSINESS LAYER.** The API layer of a service is different from the business layer. In other words, the implementation of the capabilities of a service is what we call the business layer, while the API layer is just an adapter that we place on top of the application logic in order to expose the service's capabilities to its consumers. Figure 2.3 illustrates this relationship among the layers of a service, while figure 2.4 illustrates how a user request is processed by the each of the layers.

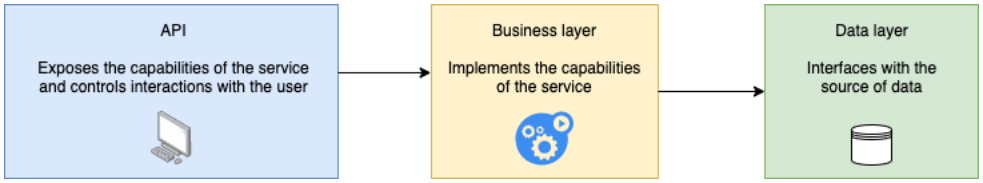


Figure 2.3 To make sure we enforce separation of concerns among the different components of our service, we structure our code around three layers: the data layer knows how to interface with the source of data used in the service; the business layer implements the service's capabilities; the interface layer implements the API which exposes the service's capabilities.

Figure 2.3 shows the high-level architecture for the orders application organized in three components: the API layer, the business layer, and the data layer. This way of structuring the application is a slight modification of the Model-View-Controller (MVC) pattern. The data layer is the part of the application that knows how to persist data so that we can retrieve it later, and it's equivalent to the Model part of the Model-View-Controller pattern. Typically, the data layer comes in the form of a collection of modules with classes which represent our data models. For example, if our persistent storage is a SQL database, the classes in the data layer will represent the tables in the database, with the help of an object relational mapper (ORM) framework. But it could also be an interface to an API which owns some of the data that we need to manipulate within the scope of our application, or to some other data source.

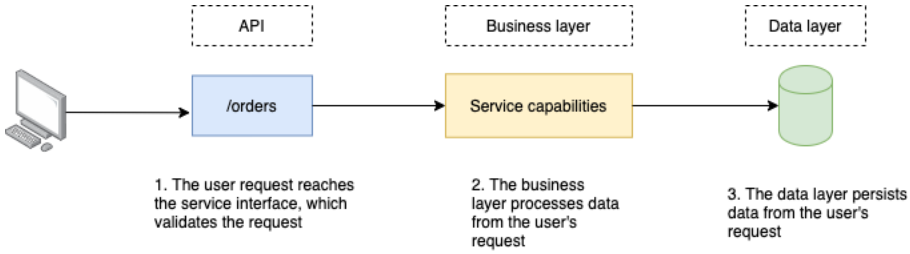


Figure 2.4 When a user request reaches the service, it's first validated by the interface layer of the service. Then the interface layer interfaces with the business layer to process the request. During processing, the business layer interfaces with the data layer in order to persist the data contained in the request.

The business layer is the part of our codebase which implements our service's capabilities. It controls the interactions between the API layer and the data layer, and in that sense, it's equivalent to the Controller part of the Model-View-Controller pattern. For the orders service, it's the part that knows what to do to place or update an order. Placing an order could be something as simple as instructing our data layer to save the order to a database, but it could involve additional tasks, such as notifying another service that an order has been placed to start its preparation, or update the user profile with their new preferences based on their latest order. The business layer orchestrates all of these actions.

The API layer is an adapter we put on top of the application layer. It implements the web API for our service. Its most important job is taking care of validating incoming requests and returning the expected responses. The API layer communicates with the business layer, passing on the data sent by the user, so that the right representation of the resources can be persisted in the server. The API layer is equivalent to the View of the Model-View-Controller pattern.

Now that we know how we are going to structure our application, let's jump straight into the code!

2.3 Implementing the API endpoints

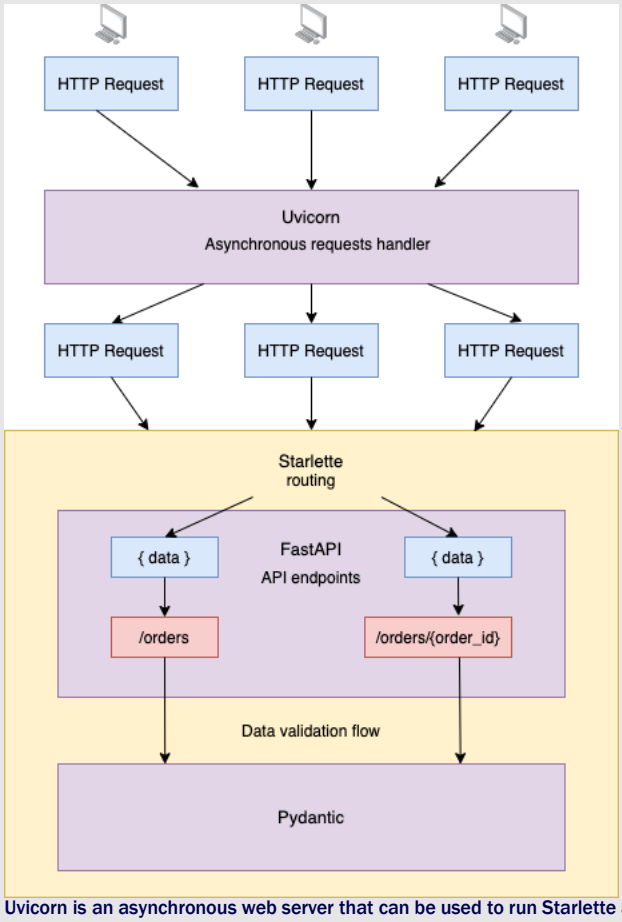
In this section, you will learn to implement the API layer of the orders service. I'll show you how to break down the implementations of the API endpoints in progressive steps. In the first step, we produce with a minimal implementation of the endpoints in which all of them return the same response. In the upcoming sections of this chapter, we'll enhance this minimal implementation to add the validation layer and the expected functionality to each endpoint. You'll also learn about the FastAPI library and how you can use it to build web API. For a brief introduction to FastAPI, head over to the sidebar titled "What is FastAPI?".

What is FastAPI?

FastAPI (https://github.com/tiangolo/fastapi) is a web API development framework built on top of Starlette (https://github.com/encode/starlette). Starlette is a high-performant, light-weight Asynchronous Server Gateway

Interface (ASGI) web framework. An ASGI means that we can implement our services as a collection of asynchronous tasks, which means that we can get important performance gains in our applications.

FastAPI applications can run with Uvicorn (https://github.com/encode/uvicorn), which is an ASGI server implementation which can be used to run asynchronous applications. In addition, FastAPI uses the fantastic data validation library Pydantic (https://github.com/samuelcolvin/pydantic/) for schema validations. Last but not least, FastAPI offers good support for OpenAPI 3, has excellent documentation, and counts with a growing community of users. The figure in this sidebar illustrates how all these different technologies fit together.



Uvicorn is an asynchronous web server that can be used to run Starlette applications. Uvicorn handles HTTP requests and passes them on to Starlette. Starlette knows which function within your app to call when a request arrives in the server. FastAPI is built on top of Starlette, and it enhances Starlette's routes with data validation and API documentation functionality. To supply such functionality, FastAPI uses Pydantic models which know how to validate the payload coming with an HTTP request.

FastAPI is currently under very active development, so by the time you are reading this, a newer version is likely to be available, which may contain changes that make it incompatible with the version that I'm using in this book, or may change some of its functionality. If you want to ensure that you're using the same version that I'm using in this

book, copy the Pipfile and the Pipfile.lock files from the GitHub repository provided with this book under the ch02 folder and run `pipenv install`.

USING THE SAME DEPENDENCIES AS IN THIS BOOK. If you want to make sure that you use the exact same dependencies that I used when writing this book, you can fetch the Pipenv and Pipenv.lock files which are provided in the ch02 of the code repository for this book and run `pipenv install`.

Before we start implementing our API, the first thing we need to do is set up our environment for this project. Create a folder named `orders` and `cd` into it. We'll use `pipenv` to install and manage our dependencies (if you don't know what `pipenv` is, head over to the sidebar titled "What is pipenv?" for a brief introduction to this tool). There're other tools available for that, and if you're not very keen on `pipenv`, you're welcomed to use your favorite dependency management tool.

What is Pipenv?

Pipenv is a dependency management tool for Python which guarantees that the exact same versions of our dependencies are installed in different environments. In other words, pipenv makes it possible to create environments in a deterministic way. To accomplish that, pipenv uses a file called Pipenv.lock which contains a description of the exact package that was installed. A description of the environment that we wish to create with pipenv is provided through a file called Pipfile.

Among other things, Pipfile contains the version of Python that must be used to create the environment, or the URLs of the different pypi repositories that must be used to pull the dependencies. Pipenv also makes it easier to keep application dependencies separate from development dependencies by providing specific installation flags for each set, and specific sections in Pipfile for each of them. For example, to install pytest we'd run `pipenv install pytest --dev`. In addition to that, pipenv also provides commands that allow us to easily manage our virtual environments, such as `pipenv shell` to activate the virtual environment, or `pipenv --rm` to delete the virtual environment.

Listing 2.2 shows how to set up a virtual environment using pipenv, and how to install the required dependencies.

Listing 2.2 Create a virtual environment and install dependencies with pipenv

```
$ pipenv --three    #A

$ pipenv install fastapi uvicorn    #B

$ pipenv shell    #C
```

#A Create a virtual environment using pipenv setting the runtime to Python 3
#B Install FastAPI and Uvicorn
#C Activate the virtual environment

To get started, copy the API specification under ch02/orders/oas.yaml in the GitHub repository provided with this book within the `orders` folder as `orders/oas.yaml` and create a sub-folder named `orders/orders`, which will contain all our service implementation. Within the `orders` folder, create a file called `app.py` and sub-folder called `orders/orders/api`, and within that folder create a file called `orders/orders/api/api.py`. At this point, the project structure should look like this:

```
.
└── orders
    ├── Pipfile
    ├── Pipfile.lock
    ├── orders
    │   ├── app.py
    │   └── api
    │       └── api.py
    └── oas.yaml
```

Listing 2.3 shows how to create an instance of the FastAPI application in file `orders/orders/app.py`.

Listing 2.3 Creating an instance of the FastAPI application

```
from fastapi import FastAPI

app = FastAPI(debug=True)    #A

from orders.api import api    #B
```

#A We create an instance of the FastAPI class. This object represents our API application.
#B We import the `api` module so that our view functions can be registered at load time

The instance of the `FastAPI` class from FastAPI is an object which represents the API that we are implementing. It provides a number of decorators that we can use to decorate our view functions and classes, and in doing so it knows how to build the documentation for our API. A decorator is a function which adds additional functionality to a function or class[1]. As you'll see in listing 2.4, the functions that we implement to handle each endpoint of the orders API contain very basic functionality – they only return a data structure. FastAPI decorators are able to transform this data structure into an HTTP response; they also map our functions to a specific HTTP path or route in our server, and are able to generate documentation based on the parameters that we pass to the decorator.

The `orders/orders/api/api.py` file will contain the definitions for the endpoints of our API. In its current implementation, FastAPI only supports function-based views. This means that we'll need to implement one view function per endpoint of our API. This is in contrast with the approach that you often find in API frameworks for Flask and Django, where you are also able to create class-based views, where each class represents a URL path, and each method in the class represents an endpoint of the URL path. As we haven't implemented the application layer yet, we'll hardcode the responses for each endpoint for now. We will not be

[1] For a classic explanation of the decorator pattern, see Erich Gamma et al., *Design Patterns* (Indianapolis IN, Addison-Wesley, 1995), pp. 175-184. For a more pythonic introduction to decorators, see Luciano Ramalho, *Fluent Python* (Sebastopol, O'Reilly, 2015), pp. 189-222.

enforcing any formats at this time, either for incoming requests or for outgoing responses, as that will be the topic of the next section. For now we are just interested in laying down the structure of our endpoints.

Listing 2.4 Initial implementation the orders API endpoints with hardcoded responses

```
from uuid import UUID

from fastapi.openapi.models import Response
from starlette import status

from orders.app import app

order = {    #A
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': 'completed',
    'created': 1740493805,
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}


@app.get('/orders')    #B
def get_orders():
    return [
        order
    ]


@app.post('/orders', status_code=status.HTTP_201_CREATED)    #C
def create_order():
    return order


@app.get('/orders/{order_id}')    #D
def get_order(order_id: UUID):    #E
    return order


@app.put('/orders/{order_id}')
def update_order(order_id: UUID):
    return order


@app.delete('/orders/{order_id}', status_code=status.HTTP_204_NO_CONTENT)
def delete_order(order_id: UUID):
    return Response(status_code=HTTPStatus.NO_CONTENT.value)    #F


@app.post('/orders/{order_id}/cancel')
def cancel_order(order_id: UUID):
    return order
```

```
@app.post('/orders/{order_id}/pay')
def pay_order(order_id: UUID):
    return order
```

#A We define an order object to return in our responses
#B In order to register a route with a decorator, we simply pass the route as a parameter to the decorator
#C We can specify the status code of the response by setting the `status_code` parameter in the decorator
#D URL parameters must be specified within curly braces
#E To use the URL parameter within our view function, we specify the parameter as an argument of our function
#F The HTTP DELETE method returns empty responses. To return an empty response with the correct headers, we must use FastAPI's Response model setting its status_code parameter to HTTPStatus.NO_CONTENT.value.

Every endpoint is defined by combining an HTTP method, which is invoked directly on the application object as a decorator, and a route or URL path. The decorators provided by FastAPI take, at a minimum, one argument, which is the route with which we want to register the decorator. Our view functions can take any number of parameters. If the name of the parameter matches the name of a URL path parameter, FastAPI will pass the path parameter from the URL to our view function on invocation. For example, as you can see in figure 2.5, the URL `/orders/{order_id}` defines a path parameter named `order_id`, and accordingly our view functions registered for that URL path take an argument named `order_id`. If a user hits the URL `/orders/53e80ed2-b9d6-4c3b-b549-258aaaef9533`, our view functions will be called with the `order_id` parameter set to `53e80ed2-b9d6-4c3b-b549-258aaaef9533`. Also note that FastAPI allows us to specify the type and format of the URL path by using type hints. In this case, our view functions specify that the format of the `order_id` parameter must be of type string with the format of a Universally Unique Identifier (UUID), so FastAPI will invalidate any calls where `order_id` doesn't follow that format.

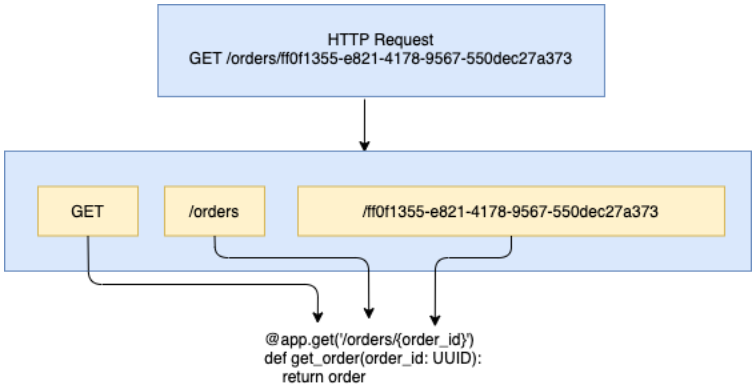


Figure 2.5 When a request arrives in the server, FastAPI knows how to map the request to the right function, and pass any relevant parameters from the request to the function. In this illustration, a GET request on the `/orders/{order_id}` endpoint with `order_id` set to `ff0f1355-e821-4178-9567-550dec27a373` is passed to the `get_order()` function

The decorators for our view functions can also take a parameter named `status_code`, which we can use to specify the status code that should be returned in the response. We are making use of this parameter in the POST `/orders` endpoint and in the DELETE `/orders/{order_id}` endpoint. For the POST endpoint, we are instructing FastAPI to return responses with the 201 status code (created), while for the DELETE endpoint we are instructing FastAPI to return a 204 status code (no content). We choose these status codes because they help us create meaningful responses in the context of the semantics of those HTTP methods. All other endpoints return a 200 status code (OK), which is the default status code returned by FastAPI. For a more detailed explanation of status codes, see section 4.3.2.

At this stage, we are returning a hardcoded response payload with the same order object in all endpoints except for the DELETE endpoint of the `/orders/{order_id}` URL path, since DELETE endpoints usually don't need to return payloads (see section 4.3.2 for more on this point). To return an empty response, we need to build a `Response` object setting the `status_code` parameter to `HTTPStatus.NO_CONTENT.value`.

At this point you can run the app to get a feeling of what the API looks like. To get the application running, run the following command from the `orders` directory:

```
$ uvicorn orders.app:app --reload
```

This will load the server for the application in development mode with hot reloading enabled. Running the server in development mode allows you to capture additional logging details, while hot reloading makes sure that, whenever you make an additional change to your files, the application automatically picks them up and reloads without you having to restart it.

You can now navigate to the http://127.0.0.1:8000/docs URL in your browser and you will be able to see an interactive display of the documentation for the API that we are implementing, as automatically generated by FastAPI directly from our code (see Figure 2.6 for an illustration)[2]. If you visit the http://127.0.0.1:8000/openapi.json URL in your browser, you will be able to see an OpenAPI representation of the same documentation in JSON.

[2] The UI that you get under http://127.0.0.1:8000/docs is known as Swagger UI, and it's one of the most popular ways of styling interactive UIs for REST APIs. Another popular style is Redoc, which is also supported by FastAPI and you can visualize under http://127.0.0.1:8000/redocs.

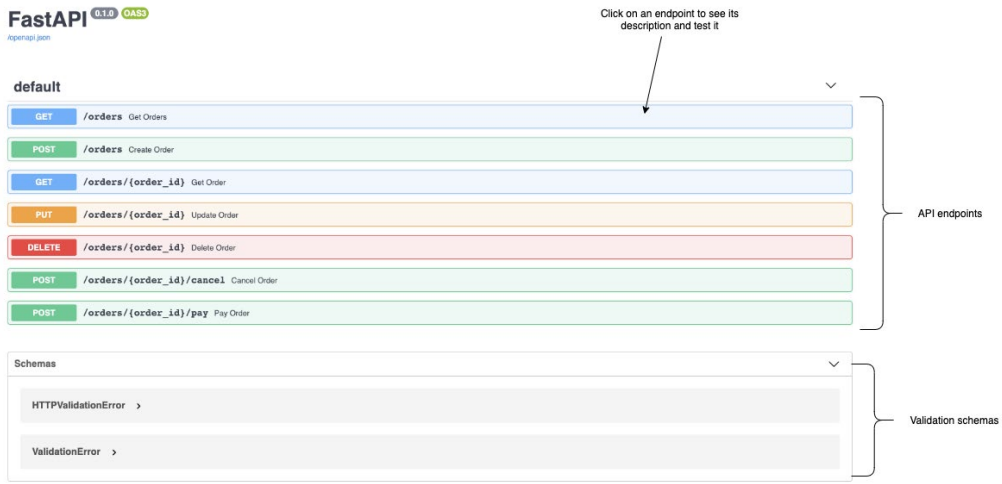


Figure 2.6 View of the Swagger UI dynamically generated by FastAPI from our code. We can use this view to test the implementation of our endpoints.

If you click in any of the endpoints represented in the Swagger UI, you will see additional documentation about the endpoint, and crucially, you will see a "Try it out" button, which gives you the opportunity to test the endpoint directly from this UI. Click on that button, and then click on "Execute", and you will get the hardcoded response that we included in our endpoints (see Figure 2.7 for an illustration).

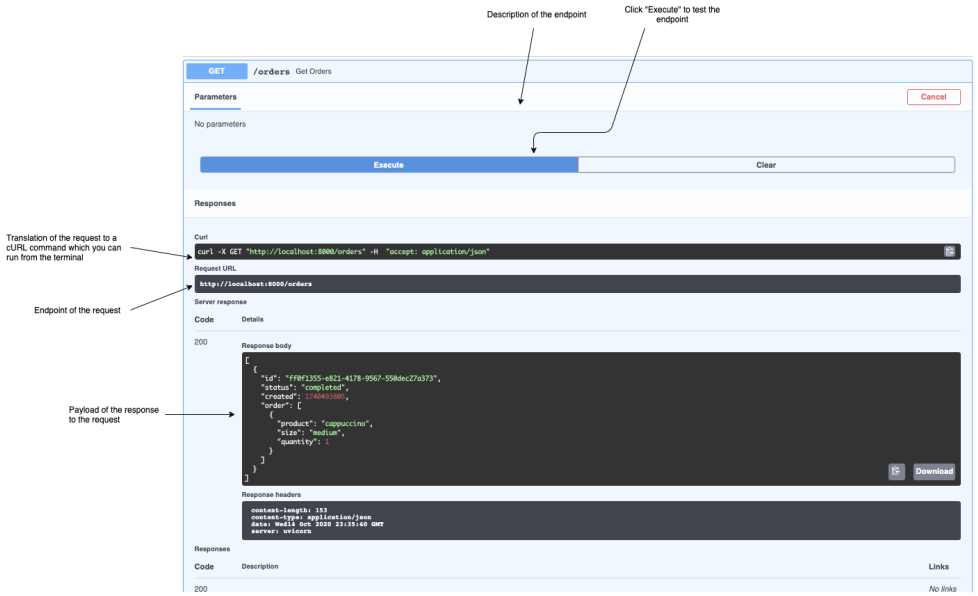


Figure 2.7 Example testing the GET /orders endpoint. To test the endpoint, click on the endpoint to expand its documentation. You'll see a "Try it out" button on the top right corner of the endpoint description. Click on that button, and then click on the execute button. This will trigger a request to the server and you'll be able to see the response sent back by the server.

Before we continue, I would like to make one clarification. The attentive reader will have noticed that, originally, we placed our API documentation together with the code under `orders/oas.yaml`. Why did we do this when FastAPI is perfectly capable of dynamically generating the documentation directly from our code? We do this is because the provided API specification is the sole source of truth about how the API should work, while the documentation dynamically generated by FastAPI is a demonstration of how your implementation works. If your implementation is right, the documentation generated by FastAPI should be nearly identical to the specification that lives under `orders/oas.yaml`.

> **THE API SPECIFICATION IS THE SOURCE OF TRUTH.** Frameworks like FastAPI can generate API documentation dynamically from our code. This documentation is a demonstration of our implementation and shouldn't be used as a reference for the API. Instead, when building an API, we should produce the specification first, and we should validate the documentation dynamically generated by the framework against the speficiation. If the implementation is correct, the documentation generated by FastAPI will be very similar to the specification.

Now that we have the basic skeleton of our API in the form of URL paths, we should move on to implementing validators for our incoming payloads and our outgoing responses. The

next section walks you through the steps needed to accomplish that, so without further ado, let's move on to that!

2.4 Validating API requests payloads and marshalling responses

Now that we have implemented the main layout for the URL paths supported by our API, this section explains how we can implement validation for incoming payloads and how we can correctly marshal our outgoing responses.

> **DEFINITION** Marshalling is the process of transforming an in-memory data structure into a format which is suitable for storage or transmission over a network. In the context of web APIs, marshalling refers more specifically to the process of transforming an object into a data structure that can be serialized into JSON, with explicit mappings for the attributes of the object and how they must be represented (see figure 2.8 for an illustration).

In my experience, and probably yours as well, this is the point where most API implementations fail to survive first contact with their clients, so it pays to invest a bit of time designing a robust approach to these issues.

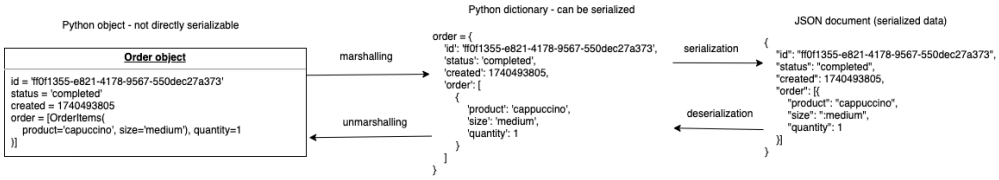


Figure 2.8 To build the payload for a response from a Python object, we first have to marshal the object into a serializable data structure, with explicit mapping of attributes from the object to the new structure. Deserializing and unmarshalling the payload should give us back an object identical to the one we serialized before.

2.4.1 Defining validation schemas with Pydantic

In this section, we learn to use Pydantic to create validation rules for our API payloads. FastAPI uses Pydantic to validate our schemas. Pydantic is a robust library that can be used to perform data validation and settings management in any Python application. Let's start by creating a new file called `orders/orders/api/schemas.py`. This file will contain the implementation of our validation models using Pydantic. Our API specification under `orders/oas.yaml` contains three schemas, namely `CreateOrderSchema`, `GetOrderSchema`, and `OrderItemSchema` (for further clarification about the location of these definitions within the API specification document, see figure 2.9).

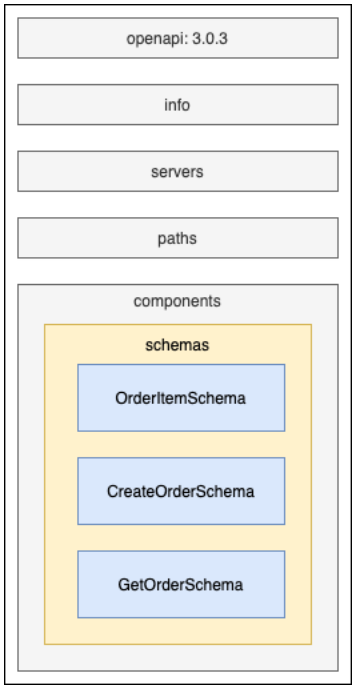


Figure 2.9 In OpenAPI 3, an API specification document contains several sections, such as the `info` section, which contains generic information about the API, the `paths` section, which contains a description of the endpoints, and the `components` section. Within the components section we find the an entry for `schemas`, and within the scope of schemas we find all the model definitions which are relevant for the API, such as `OrderItemSchema` in our case.

The specification for these schemas contains all the information we need to know to validate our payloads correctly. Let's briefly discuss the specification for `CreateOrderSchema`, `GetOrderSchema` and `OrderItemSchema` to make sure we understand how we need to implement our validation models. Listing 2.5 shows the specification for these schemas.

Listing 2.5 API specification for the schemas GetOrderSchema, CreateOrderSchema, and OrderItem

```
components:    #A
  schemas:
    OrderItemSchema:    #B
      type: object    #C
      required:    #D
        - product
        - size
      properties:    #E
        product:
          type: string
        size:
```

```
        type: string
        enum:    #F
          - small
          - medium
          - big
      quantity:
        type: integer
        default: 1    #G
        minimum: 1    #H

  CreateOrderSchema:
    type: object
    required:
      - order
    properties:
      order:
        type: array
        items:    #I
          $ref: '#/components/schemas/OrderItemSchema'    #J

  GetOrderSchema:
    type: object
    required:
      - order
      - id
      - created
      - status
    properties:
      id:
        type: string
        format: uuid
      created:
        type: integer
        description: Date in the form of UNIX timestmap    #K
      status:
        type: string
        enum:
          - created
          - progress
          - cancelled
          - dispatched
          - delivered        order:
        type: array
        items:
          $ref: '#/components/schemas/OrderItemSchema'
```

#A In OpenAPI 3, schema definitions come under the components section of the specification
#B Every schema model is a collection of key-pair values, where the key is the name of the model (`OrderItemSchema` in this case), and the values are the properties that define the schema for this model
#C Every schema has a type, which in this case is an object. An object is a collection of key-pair values
#D By default, schemas don't enforce the presence of any properties. If we want to ensure that some properties will always be present, we have to list them under the required keyword
#E The properties or attributes of the object are listed under the properties keyword
#F If a certain attribute can only take on a certain array of values, we define those values as an enumeration
#G Attributes can have a default value as in this case. Default values are normally specified for optional properties
#H We can also specify a minimum value for a property
#I If a property is an array, we must specify the type of the items in the array using the `items` keyword

#J When the type of a property is a schema defined within the same document, we can use a JSON pointer to point to that schema

#K It's also possible to provide a description that further clarifies what the property must look like

`GetOrderSchema` and `CreateOrderSchema` are two different definitions of the Order model, which are used in different contexts: `GetOrderSchema` is used when we return from the server the details of an order, while `CreateOrderSchema` is used to validate the details of an order sent by the user to the server. Since the representation of an order is different depending on whether we are sending it from the server or whether the user is sending it from a client application, we've chosen to provide to different schema definitions for each case.

Figure 2.10 illustrates how the data validation flow works for `CreateOrderSchema`. As you can see in listing 2.5 and in figure 2.10, `CreateOrderSchema` only requires the presence of one property in the payload: the `order` property, which is an array of objects whose specification is defined by `OrderItemSchema`. `OrderItemSchema` has two required properties, `product` and `size`, and one optional property: `quantity`, which has a default value of 1. This means that, when processing a request payload, we must check and validate that the `product` and `size` properties are present in the payload. Because the quantity property is optional, the API client is free to set the property in the payload or to omit it. Figure 2.10 shows what happens when the the `quantity` property is missing from the payload. In that case, we set the property to its default value of 1 in the server. For the required properties, we validate that they have the right type and come in the expected format.

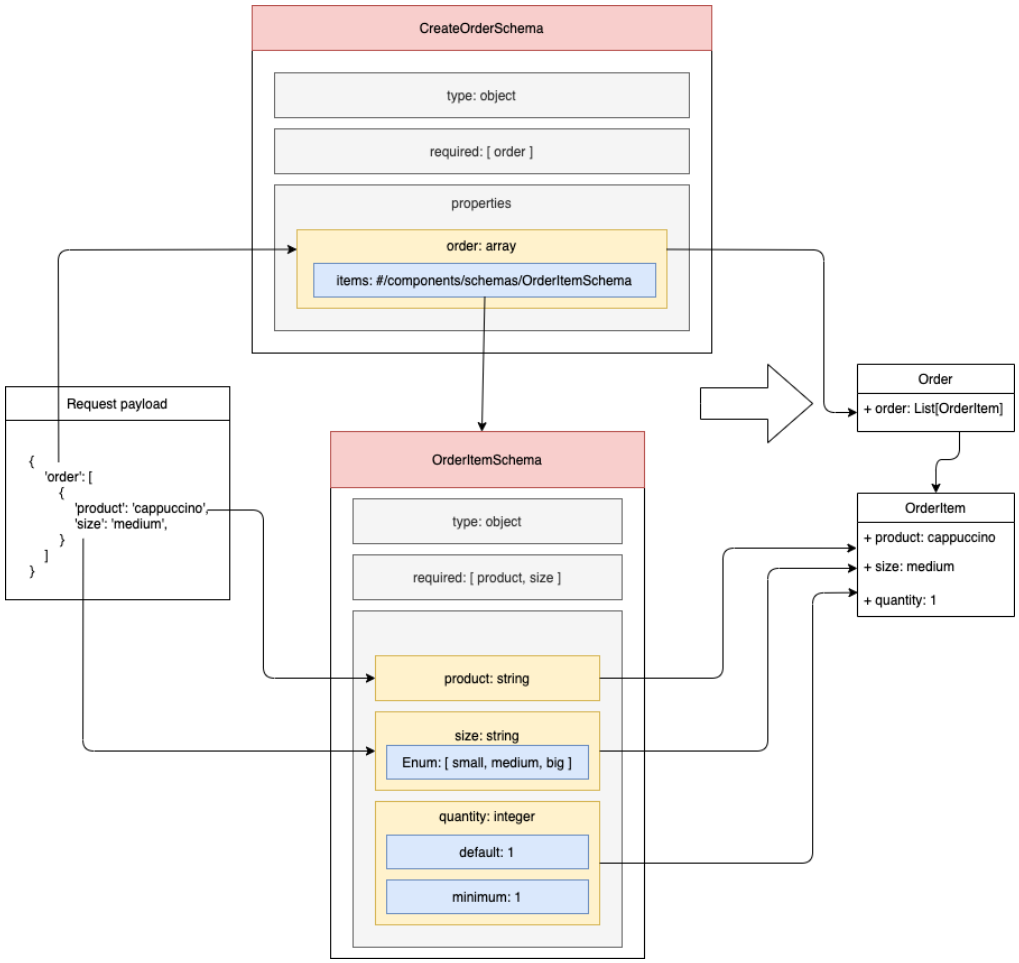


Figure 2.10 Data validation flow for request payloads against the CreateOrderSchema model. The diagram shows how each property of the request payload is validated against the properties defined in the schema, and how we build an object from the resulting validation.

What does this all look like in code? In order to define a schema in Pydantic, we simply need to implement a class which inherits from the `BaseModel` class provided by Pydantic, and define the properties of the schema as class attributes. The type and format of each attribute are defined using Python's type hints syntax. Pydantic offers a rich API (in this case, in the sense of library interface!) which allows us to customize our model definitions in multiple ways and to export them into different formats.

Listing 2.6 shows the implementation for `CreateOrderSchema`, `GetOrderSchema`, and `OrderItemSchema` in the `orders/orders/api /schemas.py` module. Every schema is defined as a class which inherits from the `BaseModel` class provided by Pydantic, and every attribute

receives a type specification using Python type hints. Where custom validation rules are needed, as in the case of the `quantity` property of `OrderItemSchema`, we use the `Field` class of the Pydantic library. For attributes that can only take on a limited selection of values, we need to define an enumeration class. The enumeration must inherit from the right type for the attribute and from the `Enum` class. In this case, we are defining enumerations for the `size` and the `status` properties.

Listing 2.6 Implementation of the validation models required by the orders API using Pydantic

```
from enum import Enum
from typing import List
from uuid import UUID

from pydantic import BaseModel, Field


class SizeEnum(str, Enum):    #A
    small = 'small'
    medium = 'medium'
    big = 'big'


class StatusEnum(str, Enum):
    created = 'created'
    progress = 'progress'
    cancelled = 'cancelled'
    dispatched = 'dispatched'
    delivered = 'delivered'

class OrderItemSchema(BaseModel):    #B
    product: str    #C
    size: SizeEnum    #D
    quantity: int = Field(default=1, ge=1)    #E


class CreateOrderSchema(BaseModel):
    order: List[OrderItemSchema]


class GetOrderSchema(CreateOrderSchema):
    id: UUID    #F
    created: int = Field(description='Date in the form of UNIX timestmap')    #G
    status: StatusEnum
```

#A This class defines an enumeration schema whose values are all strings
#B Every schema model must inherit from `pydantic.BaseModel`
#C We use Python type hints to specify the type of an attribute
#D When an attribute can only take on a certain array of values, we set its type to an enumeration
#E We can specify additional validation rules with the `Field` class. Here we use it to specify the default value of the attribute and its minimum value
#F In OpenAPI, a UUID is just a string with UUID format. By setting the type of the `id` attribute to UUID, Pydantic is able to infer that this attribute must be a string with UUID format
#G When needed, we can also pass a description parameter to the `Field` class, which will be used by FastAPI in the dynamically generated documentation of the API

Now that our validation models are implemented, it's time to hook them up with the API and use them to validate and marshal payloads!

2.4.2 Validating request payloads with Pydantic

In this section, we use the models implemented in the previous section to validate request payloads in the API. How do we access request payloads within our view functions? In FastAPI, request payloads can be intercepted within our view functions by declaring them as a parameter of the function, and in order to validate them, we simply need to set their type to the relevant Pydantic model, as shown in listing 2.7 (we've highlighted the relevant changes with bold characters and omitted irrelevant parts of the code with an ellipsis).

Listing 2.7 Hooking validation models up with the API endpoints

```
from typing import List
from uuid import UUID

from fastapi.openapi.models import Response
from starlette import status

from orders.app import app
from orders.api.schemas import GetOrderSchema    #A


...


@app.post('/orders', status_code=status.HTTP_201_CREATED)
def create_order(order_details: CreateOrderSchema):    #B
    return order


@app.get('/orders/{order_id}')
def get_order(order_id: UUID):
    return order


@app.put('/orders/{order_id}')
def update_order(order_id: UUID, order_details: CreateOrderSchema):
    return order

...
```

#A We import the Pydantic models so that we can use them to hook them into FastAPI for validation
#B When an endpoint accepts request payloads, we capture the payload by specifying a parameter in our function signature which represents the payload. In order to make FastAPI validate the payload, we use type hints setting the type of the payload to the Pydantic model that we want to use to validate it

If you kept the application running, the changes will be loaded automatically by the server, so you just need to refresh the browser in order for the UI to update. If you click on the POST endpoint of the /orders URL path, you'll see that the UI now gives you an example of the payload expected by the server. You can try out sending a request with the sample payload, and it'll succeed. Now, if you try editing the payload to remove any of the required

fields, for example the product field, and execute it again, you'll get the following error message:

```
{
  "detail": [
    {
      "loc": [
        "body",
        "order",
        0,
        "product"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```

FastAPI generates an error message which points to where in the payload the error is found. The error message uses a JSON pointer to indicate where the problem is. A JSON pointer is a specific syntax in JSON Schema that allows you to represent the path to a specific value within a JSON document. If this is the first time that you encounter JSON pointers, to make it easier to understand them, think of them as a different way of representing dictionary syntax and index notation in Python. For example, the following error message: "`loc: /body/order/0/product`" is roughly equivalent to the following notation in Python: `loc['body']['order'][0]['product']`. Figure 2.11 shows you how to interpret the JSON pointer from the error message to identify the source of problem in the payload.

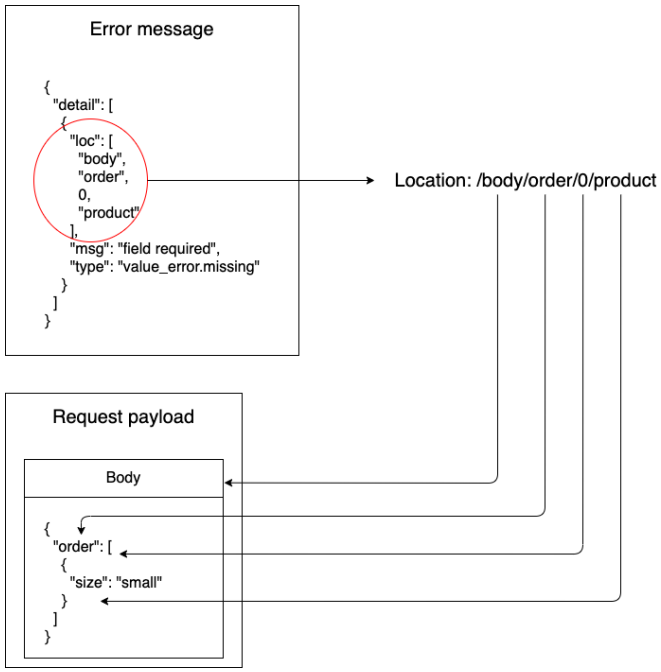


Figure 2.11 When a request fails due a malformed payload, we'll get a response with an error message. The error message uses a JSON pointer to tell us where the error is. In this case, the error message says that the property /body/order/0/product is missing from the payload.

You can also try to change the payload so that, instead of missing a required property, it contains an illegal value for the size property, for example:

```
{
  "order": [
    {
      "product": "string",
      "size": "somethingelse"
    }
  ]
}
```

In this case, you'll also get an informative error with the following message: "value is not a valid enumeration member; permitted: 'small', 'medium', 'big'". What happens if we include an invalid or illegal property, that is a property which hasn't been defined in the schema, in the payload? For example:

```
{
  "order": [
    {
      "product": "string",
      "size": "small",
      "invalid": "invalid"
```

```
    }
  ]
}
```

In this case, FastAPI will happily accept the payload, and just drop the invalid properties before passing the data to our view functions. This is the default behavior in JSON Schema. By default, JSON Schema doesn't invalidate payloads which come with unknown attributes. If we want to invalidate unknown attributes, we have to explicitly set the schema to reject additional properties. We'll learn about this Chapter 6. In my experience, most APIs are not configured to raise errors when a client sends unknown properties. In some cases, this is probably fine. However, in some situations, lack of validation for illegal properties can lead to confusion between the client and the server, especially if the client makes a typo in an optional property. For example, if a client sent the following payload, with a typo in the `quantity` property:

```
{
  "order": [
    {
      "product": "string",
      "size": "small",
      "quantit": 5
    }
  ]
}
```

FastAPI would drop the `quantit` property and assume the client wishes to set the quantity property to 1. This situation could lead to confusion between the client and the server, and in such cases, validating for illegal properties might help to create a more meaningful communication between the client and the server. In Chapter 6, we'll see how to implement such validations.

If you set the quantity property to 0, you'll also get an error response with the following message: "ensure this value is greater than or equal to 1".

As you may have noticed, the default payload example provided by FastAPI is not ideal:

```
{
  "order": [
    {
      "product": "string",
      "quantity": 0,
      "size": "small"
    }
  ]
}
```

In this sample payload, the value of `quantity` is set to 0, which is an illegal value. We can overcome this limitation by providing an example value for this property in the model definition:

```
class OrderItemSchema(BaseModel):
    product: str
    size: SizeEnum
    quantity: int = Field(default=1, ge=1, example=1)
```

If you refresh the browser now, the sample payload will look correct:

```
{
  "order": [
    {
      "product": "string",
      "quantity": 1,
      "size": "small"
    }
  ]
}
```

Now that we know how to validate request payloads, let's see how we can use our Pydantic models to marshal and validate our requests.

2.4.3 Marshalling and validating response payloads with Pydantic

In this section, we'll use the Pydantic models implemented ealier to marshal and validate the response payloads of our API. It's important that we validate our responses payloads, since malformed payloads will break the integration with our API consumers. For example, the schema for the response payload of the POST `/orders` endpoint is `GetOrderSchema`, which requires the presence of the `id`, `created`, `status`, and `order` fields. API clients will expect the presence of all these fields in the response payload, and will raise errors if any of the fields is missing or comes in the wrong type or format. In fact, malformed response payloads are one of the most common causes of API integration failures.

> **VALIDATE RESPONSE PAYLOADS.** Malformed response payloads are a very common source of API integration failures. You can avoid this problem by validating your response payloads bvefore they leave the server. In FastPAI, this is easily done by setting the `response_model` parameter of a view function decorator. Beware, if you have very large response payloads (as is common in data-driven applications), it may be costly to validate the whole payload. In those cases, you may want to validate just a representative part of the data.

To validate responses, we simply need to pass the models to FastAPI's decorators through the `response_model` parameter, as shown in Listing 2.8.

Listing 2.8 Hooking validation models for responses in the API endpoints

```
from typing import List
from uuid import UUID

from fastapi.openapi.models import Response
from starlette import status

from orders.app import app
from orders.api.schemas import GetOrderSchema, CreateOrderSchema    #A


...
```

```
@app.get('/orders', response_model=List[GetOrderSchema])    #A
def get_orders():
    return [
        order
    ]


@app.post('/orders', status_code=status.HTTP_201_CREATED, response_model=GetOrderSchema)
def create_order(order_details: CreateOrderSchema):
    return order


@app.get('/orders/{order_id}', response_model=GetOrderSchema)
def get_order(order_id: UUID):
    return order


@app.put('/orders/{order_id}', response_model=GetOrderSchema)
def update_order(order_id: UUID, order_details: CreateOrderSchema):
    return order


@app.delete('/orders/{order_id}')
def delete_order(order_id: UUID):
    return Response(status_code=HTTPStatus.NO_CONTENT.value)


@app.post('/orders/{order_id}/cancel', response_model=GetOrderSchema)
def cancel_order(order_id: UUID):
    return order


@app.post('/orders/{order_id}/pay', response_model=GetOrderSchema)
def pay_order(order_id: UUID):
    return order
```

#A We let FastAPI know which model to marshal and validate against by passing it to the decorator using the `response_model` parameter

Generally, FastAPI raises an error if a required property is missing from our response payload. It also removes any properties which are not part of the schema, and it tries to cast each property into the right type. Let's see this behavior at work.

If you inspect the GET endpoint for the `/orders` URL path and try it out, you'll get without issues the order that we hardcoded earlier at the top of the `orders/orders/api/api.py` file. Let's make some modifications to that payload to see how FastAPI handles them. To begin with, let's add one additional property to the payload:

```
# orders/orders/api/api.py
...

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': 'completed',
    'created': 1740493805,
    'updated': 1740493905,
    'order': [
```

```
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 0
        }
    ]
}

...
```

If we try out the GET endpoint of the `/orders` URL path again, we'll get the same response we obtained before, without the updated property that we just added:

```
[
  {
    "order": [
      {
        "product": "cappuccino",
        "size": "medium",
        "quantity": 1
      }
    ],
    "id": "ff0f1355-e821-4178-9567-550dec27a373",
    "created": 1740493805,
    "status": "completed"
  }
]
```

Let's now try to remove the created property from the order payload:

```
# orders/orders/api/api.py
...

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': 'completed',
    'updated': 1740493905,
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}

...
```

In this case, FastAPI raises a server error telling us that the required `created` property is missing from the payload:

```
pydantic.error_wrappers.ValidationError: 1 validation error for GetOrderSchema
response -> 0 -> created
  field required (type=value_error.missing)
```

Let's now change the value of the created property to a string:

```
# orders/orders/api/api.py
```

```
...

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': 'completed',
    'created': '1740493805',
    'updated': 1740493905,
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}

...
```

In this case, FastAPI will cast the string value of the `created` property to an integer and send the expected payload. In Chapter 6, we'll discuss the potential drawback of this lack of strong type checking.

If we set the value of the `created` property to a string representing a value that cannot be cast into an integer, FastAPI will raise a helpful error:

```
pydantic.error_wrappers.ValidationError: 1 validation error for GetOrderSchema
response -> 0 -> created
  value is not a valid integer (type=type_error.integer)
```

In addition to validating that our properties have the expected data types, we'd also like to validate that they have the expected values. For example, the `quantity` property must have a minimum value of 1, and the `status` property can only have one of the following five values: created, progress, or cancelled, dispatched, or delivered. Change our hardcoded order payload with the following values:

```
# orders/orders/api/api.py
...

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': 'completed',
    'created': '1740493805',
    'updated': 1740493905,
    'order': [
        {
            'product': 'cappuccino',
            'size': invalid,
            'quantity': 0
        }
    ]
}

...
```

If you cal the GET /orders endpoint, you'll see that FastAPI raises helpful errors in the terminal invalidating the properties of this payload.

Now that we have seen how to validate request payloads and marshal our responses, let's add a simple state management mechanism for the application so that we can add orders and change their state through the API.

2.5 Adding an in-memory list of orders to the API

So far, our API implementation has been returning always the same response object. We haven't been able to add orders and play around with them. Let's change that by adding a simple in-memory collection of orders to manage the state of the application. We'll represent the collection of orders as a Python list. We'll manage the list within the view functions of the API layer. The implementation that we offer in this section is very simple and only intended for illustration purposes. In particular, this implementation doesn't include error handling, and we will also skip the implementation of the data/model and the application/controller layers in order to keep things simple. Since we won't have a data layer, some of the functionality that we'd normally delegate to the data layer (such as creating IDs) will be handled by the API. In Chapter 7, we'll add the controller layer and the data layer by learning useful service implementation patterns.

Listing 2.9 contains the changes required for the view functions under in `api.py` to manage the in-memory list of orders. The collection of orders is represented by a simple Python list assigned to the variable `ORDERS`. In this list, we store the details of every ordered in dictionary form. Any updates to the order are performed directly on the dictionary. We could use objects to do this and enhance them with methods that encapsulate each of the operations that we perform on the orders, such as paying or cancelling, but we've opted for a simpler approach to keep the code focused on the API layer.

Listing 2.9 Managing the application's state with an in-memory list

```
import time    #A
import uuid
from http import HTTPStatus
from typing import List
from uuid import UUID

from fastapi import HTTPException
from fastapi.openapi.models import Response
from starlette import status

from orders.app import app
from orders.api.schemas import GetOrderSchema, CreateOrderSchema


ORDERS = []    #B


@app.get('/orders', response_model=List[GetOrderSchema])
def get_orders():
    return ORDERS    #C


@app.post('/orders', status_code=status.HTTP_201_CREATED, response_model=GetOrderSchema)
def create_order(order_details: CreateOrderSchema):
```

```
    order = order_details.dict()    #D
    order['id'] = uuid.uuid4()    #E
    order['created'] = time.time()
    order['status'] = 'created'
    ORDERS.append(order)    #F
    return order    #G


@app.get('/orders/{order_id}', response_model=GetOrderSchema)
def get_order(order_id: UUID):
    for order in ORDERS:    #H
        if order['id'] == order_id:
            return order
    raise HTTPException(    #I
        status_code=404, detail=f'Order with ID {order_id} not found'
    )


@app.put('/orders/{order_id}', response_model=GetOrderSchema)
def update_order(order_id: UUID, order_details: CreateOrderSchema):
    for order in ORDERS:
        if order['id'] == order_id:
            order.update(order_details.dict())
            return order
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )


@app.delete('/orders/{order_id}')
def delete_order(order_id: UUID):
    for index, order in enumerate(ORDERS):    #J
        if order['id'] == order_id:
            ORDERS.pop(index)
            Response(status_code=HTTPStatus.NO_CONTENT.value)
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )


@app.post('/orders/{order_id}/cancel', response_model=GetOrderSchema)
def cancel_order(order_id: UUID):
    for order in ORDERS:
        if order['id'] == order_id:
            order['status'] = 'cancelled'
            return order
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )


@app.post('/orders/{order_id}/pay', response_model=GetOrderSchema)
def pay_order(order_id: UUID):
    for order in ORDERS:
        if order['id'] == order_id:
            order['status'] = 'progress'
            return order
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
```

```
    )
```

#A To implement the new functionality, we need to import new utilities into our API module
#B We represent our in-memory list of orders with a Python list assigned to the variable ORDERS
#C To return the list of orders, we simply return the ORDERS list
#D To capture the details of an order, we first turn it into a dictionary
#E We enhance the order object with information that must be set in the server on creation, such as the creation date and its status
#F To create the order, we simply add it to the list
#G After appending the order to the list, we return it
#H To find an order by ID, we simply iterate the ORDERS list and check the ID of each order against the ID requested by the user
#I If an order isn't found, we raise an HTTPException with status_code 404, which instructs FastAPI to return a 404 response to the user
#J To delete an order from the list, we iterate the list using the enumerate function, which returns the index of each order in the list. We then use the index to remove the order from the list using the list.pop() method

If you play around with the POST /orders endpoint, you'll be able to create new orders, and using their IDs you'll be able to alter their states by hitting the UPDATE `/orders/<order_id>` endpoint, or the POST `/orders/<order_id>/pay` endpoint, for example. In every endpoint under the `/orders/{order_id}` URL path, we check whether the order requested by the API client exists, and if it doesn't, we return a 404 (Not Found) response with a helpful message.

At this point, we are now able to use the orders service to create orders, update them, pay for them, cancel them, and get their details. You have now implemented a fully working web API for a microservice application. You have learned to structure the application in a way that reduces the risk of tight coupling among components, learned a bunch of new libraries to build web APIs, and learned to add robust data validation mechanisms to your APIs. Most of all, you've learned to put it all together and run it with success. That's no small feat, and I congratulate you on that! Hopefully this chapter has sparked your interest and excitement about what's possible when you build microservices exposing web APIs. In the ensuing chapters, we'll delve deeper into the topics, concepts, and ideas explored in this chapter!

2.6 Summary

- When implementing a microservice, you need to avoid tight coupling among its components. You can accomplish this by applying a variation of the Model-View-Controller pattern. According to this pattern, you can structure your application into three layers:
- A data layer which knows how to interface with the source of data (the Model)
- An controller layer which implements the capabilities of the service (the Controller)
- An interface layer which exposes the capabilities of the service through an API (the View)
- FastAPI is a popular web API framework for Python. It's highly performant, has excellent documentation and counts with an increasing community of users. It makes an excellent choice for the implementation of microservice web APIs.
- FastAPI uses Pydantic for data validation. Pydantic is a popular and highly performant data validation library which uses type hints to create validation rules. These features

make it very intuitive and easy to use.

- When implementing an API, you can first lay out the endpoints with hardcoded responses and then move on to implementing the validation models.
- You must validate request payloads in the API layer before you pass the data on to the controller layer to ensure the data arrives in the expected format.
- You must validate response payloads at the API layer before they leave the server to ensure they're sent in the expected format and avoid integration errors with the API consumers.
- You can use the UI interface dynamically generated by FastAPI to test the endpoints that you have implemented, to understand the behavior of your implementation, and discover issues with your implementation.

3

Designing and managing microservice APIs

This chapter covers

- Types of web APIs, and protocols and their main features
- Defining the application boundaries
- Strategies for decomposing an application into microservices
- Managing APIs using the principles of API governance
- Using documentation-driven development to improve the API development process

We can use a great variety of protocols and technologies to implement our microservices web APIs. It is important that we know and understand what our choices are so that we can make the best decisions for our APIs. In this chapter, we begin by studying the main types of web APIs and protocols that we can use to implement web interfaces for our microservices, and we examine their advantages and constraints.

We also discuss two different techniques that we can use to decompose our system into microservices: decomposition by business capability and decomposition by subdomains. Service decomposition is a fundamental step in the design of our microservices strategy, since it helps us define applications with clear boundaries, well-defined scopes, and explicit responsibilities. A well-designed microservices architecture is essential to reduce the risk of ending up with a distributed monolith instead. We will see how the concepts of Domain Driven Design (DDD) can help us in this process, and what elements we need to take into account when defining the scope of a service.

We also introduce the concept of API Governance. API Governance refers to several best practices in API management that help organizations deliver solid and stable APIs. We describe the most important aspects of API Governance when it comes to managing the lifecycle of our APIs, their evolution over time, and the application of different layers of security.

Finally, we introduce the concept of documention-driven development. We explain what it means, what its implications are, and how it helps us avoid many of the most common problems

that API developers face, especially when it comes to integration between API producers and consumers.

3.1 Types of web APIs and protocols

In this section, we explain the different types of API protocols that we can use to implement web application interfaces. Each of these protocols evolved to address specific problems in the design and implementation of web API interfaces, as well as in the interaction between API consumers and producers. We will discuss the benefits of each protocol in different situations, so that we can make the best choice of API protocol when implementing our own APIs. We will discuss the following protocols:

- RPC and its variants JSON-RPC and XML-RPC
- SOAP
- gRPC
- REST
- GraphQL

Choosing the right type of protocol is fundamental for the performance and integration strategy of our APIs. The factors that will condition our choice of API protocol include

- Whether the API is public or private.
- Type of API consumer: whether it is small devices, mobile applications, browsers, web servers, other application/services within our system, and so on.
- The capabilities and resources that we wish to expose; for example, whether it is a hierarchical data model which can be easily organized around a few endpoints, or a highly interconnected net of resources with cross-references among them.

We will take these factors into consideration when discussing the benefits and constrains of each protocol in the following sections in order to assess their suitability for different scenarios.

3.1.1 Web APIs in RPC, XML-RPC, JSON-RPC

This section explains what a Remote Procedure Call is, and its two most common implementations in the context of web APIs, namely XML-RPC and JSON-RPC. As you can see in figure 3.1, a Remote Procedure Call (RPC) is a protocol that allows a client to invoke a procedure or subroutine in a different machine. The origins of this form of communication go back to the 1980s, with the emergence of distributed computing systems, and over time it has evolved into a number of standard implementations of this protocol[19]. Two popular implementations for web-based APIs are XML-RPC and JSON-RPC.

[19] Bruce Jay Nelson is credited with the introduction of the term "remote procedure call" in his doctoral dissertation, available under Bruce Jay Nelson, *Remote Procedure Call* (Technical Report CSL-81-9, Xero Palo Alto Research Center, Palo Alto CA, 1981). For a more formal description of the implementation requirements of RPC, see Andrew B. Birrell and Bruce Jay Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, 2:1 (1984), pp. 39-59.

Figure 3.1. Illustration of Remote Procedure Call (RPC): a program running in Server 1 invokes a function or subroutine from a program running in Server 2.

XML-RPC is a Remote Procedure Call protocol which uses Extensible Markup Language (XML) over HTTP to exchange data between a client and a server. It was created by Dave Winer in 1998, and it eventually grew into what later came to be known as SOAP (see section 3.1.2 for a description of SOAP).

With the increasing popularity of JavaScript Object Notation (JSON) as a standard serialization format for the exchange of data across different processes and services, an alternative implementation of RPC came in the form of JSON-RPC. It was introduced in 2005 and it offers a simplified way of structuring the information which is exchanged between the client and the server. Figure 3.2 illustrates the communication process between two machines using JSON-RPC. As you can see in the image, JSON-RPC payloads usually include three properties:

- `method`: the method or function which the client wishes to invoke in the remote server.
- `params`: the parameters which must be passed to the method or function on invocation.
- `id`: a value to identify the request.

In turn, JSON-RPC response payloads include the following parameters:

- `result`: the value returned by the invoked method or function.
- `error`: an error code raised during the invocation, if any.
- `id`: the id of the request which is being handled.

RPC is a light-weight protocol which allows you to make API integrations without having to implement complex interfaces. An RPC client only needs to know the name of the function that it needs to invoke in the remote server, together with its signature. It doesn't need to look for different endpoints and comply with their payloads schemas as in REST. However, the lack of a proper interface layer between the API consumer and the producer inevitably tends to create tight coupling between the client and the specific implementation details of the server. As a consequence of this coupling, a small change in the implementation details of the functions consumed by the client risk breaking the integration, resulting in unexpected responses being sent to the client. For this reason, RPC is recommended mostly for private API integrations, where you're in full control of both the client and the server.

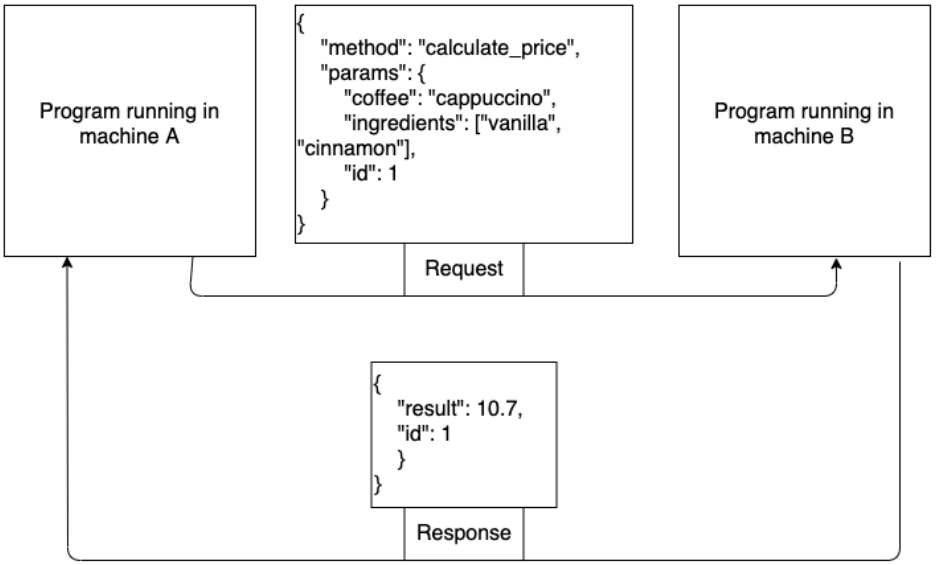


Figure 3.2 Example of JSON-RPC communication between two services running in different machines. A program running in machine A sends a request to another program running in machine B, instructing it to make the following method invocation: calculate_price(coffee='cappuccino', ingredients=['vanilla', 'cinnamon']). The program running in machine B responds with the result of the invocation: $10.7.

3.1.2 Web APIs in SOAP

This section discusses the Simple Object Access Protocol (SOAP) and how it is used in the implementation of web application interfaces. The SOAP protocol enables communication with web services through the exchange of payloads in XML. It was introduced in 1998 by Dave Winer, Don Box, Bob Atkisnon, and Mohsen Al-Ghosein for Microsoft, and after a number of iterations, it became a standard protocol for web applications in 2003. SOAP was conceived as a messaging protocol and it runs on top of a data transport layer, typically HTTP.

SOAP was designed to meet three major goals:

1. **Extensibility**: SOAP can be extended to add capabilities found in other messaging systems.
2. **Neutrality**: it can operate over any data transfer protocol of choice, including HTTP, or directly over TCP or UDP, among others.
3. **Independence**: SOAP enables communication between web applications regardless of their programming models.

The payloads exchanged with a SOAP endpoint are structured in XML, and as illustrated in figure 3.3, they include the following properties, some of which are required, and some of which are optional:

- **Envelop (required)**: it identifies the XML document as a SOAP.
- **Header (optional)**: it includes additional information about the data contained in the message, for example the type of encoding.
- **Body (required)**: it contains the payload (actual message being exchanged) of the

request/response.

- **Fault (optional)**: it contains information about errors that occurred while processing the request.

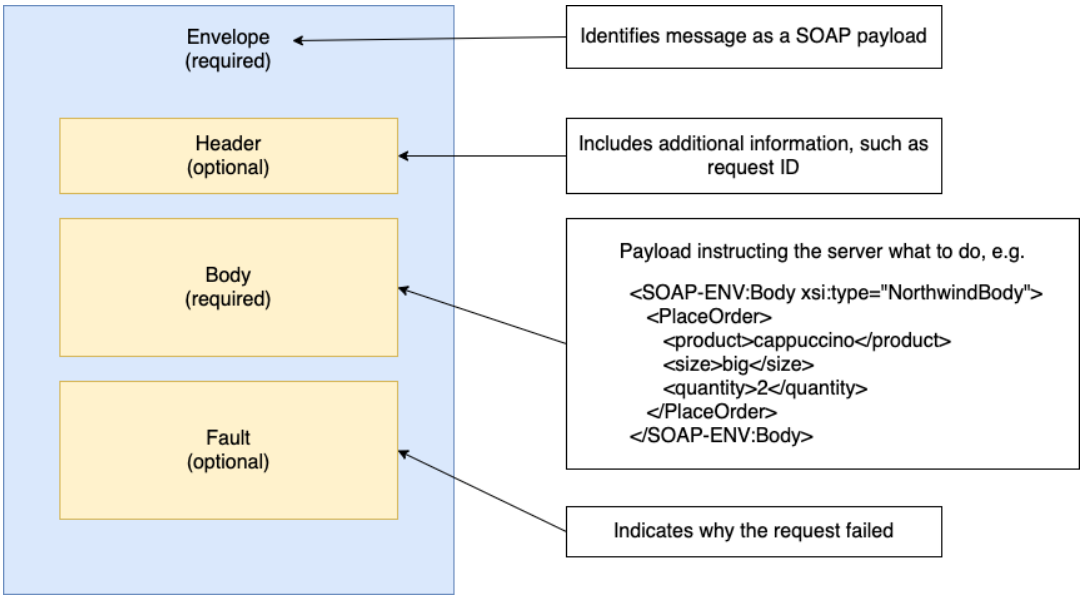


Figure 3.3 Structure of a typical SOAP message: at the top level of the message, an Envelope component tells us what type of message this is, which in this case is a SOAP payload. An optional Header component includes information about the data included in this message, such as the type of encoding. The Body component includes the actual payload of the message: the data being exchanged between the client and the server.

SOAP was a major contribution to the world of web applications. The availability of a standard protocol for communication across web applications led to the emergence of vendor APIs. Suddenly, it was possible to sell digital services by simply exposing an API that everybody could understand and interact with.

In recent years, SOAP has been superseded by newer protocols and architectures. The factors that contributed to the decline of SOAP include:

- The payloads exchanged through a SOAP interface contain large XML documents which consume a large amount of bandwidth.
- XML is difficult to read and maintain, and it requires careful parsing, which makes exchanging messages structured in XML less convenient.
- SOAP does not provide a clear framework for organizing the data and capabilities that we want to expose through an API. It provides a way of exchanging messages, and it is up to the agents involved on both sides of the API to decide how to make sense of such messages.

3.1.3 Web APIs in gRPC

This section discusses a specific implementation of the Remote Procedure Call (RPC) protocol called gRPC[20], which was developed by Google in 2015. This protocol uses HTTP/2 as a transport layer, and exchanges payloads encoded with Protocol Buffers (Protobuf). Protocol Buffers is a method for serializing structured data. Serialization, or marshalling, is the process of translating data into a format that can be stored or transferred over a network (for an explanation of how serialization works, see section 2.4 in chapter 2). Another process must be able to pick up the saved data and restore it to its original format. The process of restoring serialized data is also known as unmarshalling.

Some serialization methods are language-specific, such as `pickle` for Python. Some others, like the popular JavaScript Object Notation (JSON) format, are language-agnostic and can be translated into the native data structures of other languages, like dictionaries in the case of Python. JSON is a platform independent data interchange format, which uses human-readable text to represent data in the form of key-value pairs.

An obvious shortcoming of JSON, in comparison to `pickle`, is that it only allows for the serialization of simple data representations, consisting of strings, booleans, arrays (lists in Python), associative arrays (dictionaries in Python), and `null` (`None` in Python) values. Because JSON is language-agnostic and must be strictly transferable across languages and environments, it cannot allow for the serialization of language-specific features, like `NaN` (not a number) in JavaScript, tuples or sets in Python, or classes in object-oriented languages.

The `pickle` format allows you to serialize any type of data structure running in your Python programs, including custom objects. The shortcoming, though, is that the serialized data is highly specific to the version of Python that you were running at the time of dumping the data. Due to slight changes in the internal implementation of Python bewteen different releases, you cannot expect a different process to be able to reliably parse a pickled file, unless it runs in the exact same environment as the program which created the pickled file.

[20] You're surely wondering what the "g" in gRPC stands for. According to the official documentation, it stands for a different word in every release. For example, in version 1.1 it stands for "good", while in version 1.2 it stands for "green", and so on (https://grpc.github.io/grpc/core/md_doc_g_stands_for.html). Some people believe that the "g" stands for Google, as this protocol was invented by Google (see "Is gRPC the future of client-server communication? by Bleeding Edge Press: https://medium.com/@EdgePress/is-grpc-the-future-of-client-server-communication-b112acf9f365).

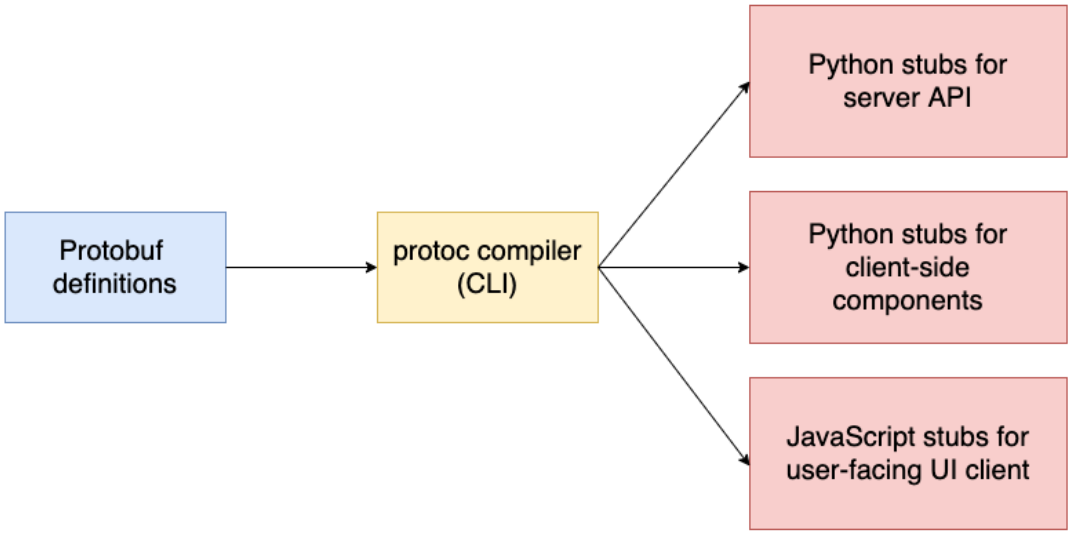


Figure 3.4 gRPC uses Protobuf to encode the data exchanged through the API. In order to integrate services exposing gRPC APIs, we first need to produce a Protobuf specification, and then use those specifications to generate code (stubs) for both the server and the client which is able to handle Protobuf payloads

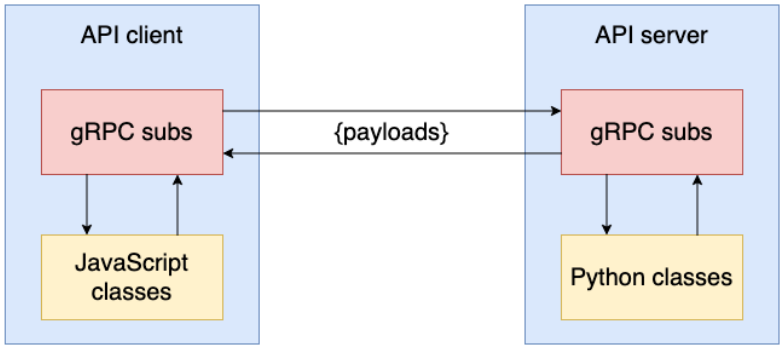


Figure 3.5 The stubs generated with Protobuf take care of parsing the payloads exchanged between the API client and the API server and translating them into native code

Protobuf comes somewhere in between: it allows you to define more complex data structures than JSON, including enumerations, and it is able to generate native classes from the serialized data, which you can extend to add custom functionality. As you can see in figure 3.4, in gRPC, you must first define the schema for the data structures that you want to exchange over the API using the Protobuf specification format, and then use the Protubuf CLI to automatically generate code for both the client and the producer of the API. The data structures generated from the Protobuf specifications are called stubs.

gRPC offers a more reliable approach for API integrations than plain RPC. The use of Protobuf serves as an enforcement mechanism which ensures that the data exchanged between the client

and the server comes in the expected format. It also helps to make sure that communication over the API is highly optimized, since the data is exchanged directly in binary format. For this reason, gRPC is an ideal candidate for the implementation of internal API integrations where performance is a relevant factor.

3.1.4 Web APIs in REST

This section explains what Representational State Transfer (REST) is and what its main features are. Representational State Transfer is an architectural style which defines a number of constrains for the design of web services. We will explore the design principles of RESTful APIs in detail in chapter 4, but suffice it to say that the main idea behind REST is to build stateless web APIs structured around resources, and to leverage the proper use of the HTTP protocol in order to build meaningful requests and responses.

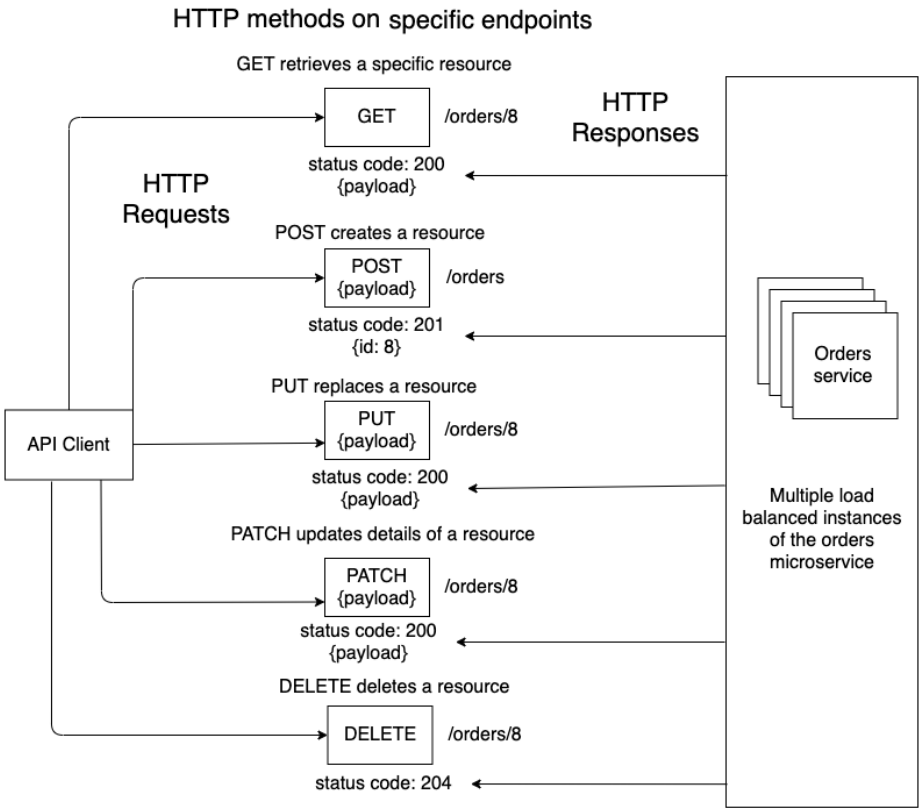


Figure 3.6 Flow diagram of a Representational State Transfer (REST) API. An API client can use different HTTP methods to perform different actions on the /orders API endpoint, such as GET to retrieve details about a specific order, or POST to create an order. On the server side, we can have any number of instances of the same service in order to load balance requests, thanks to the stateless nature of the protocol. The server responds with the expected HTTP status code and a payload. The response for a POST request includes the ID of the resource that has been created.

As illustrated in figure 3.6 and we will see further in chapter 4, the HTTP protocol includes a large number of resources which can be used to design expressive REST APIs, such as the HTTP methods, the Header fields, the concept of Uniform Resource Identifier (URI), response status codes, and the ability to send data payloads, among others.

One of the core concepts of REST is the idea of resource. We distinguish two types of resources: collections and singletons, and we use different URL paths to represent each of them. For example, in figure 3.7, the `/orders` endpoint represent a collection of orders, while the `/orders/{order_id}` endpoint represents the URI of a single order. The /orders URL path can be used to retrieve a list of orders or to place new orders, while the `/orders/{order_id}` endpoint can be used to perform specific actions on a single order.

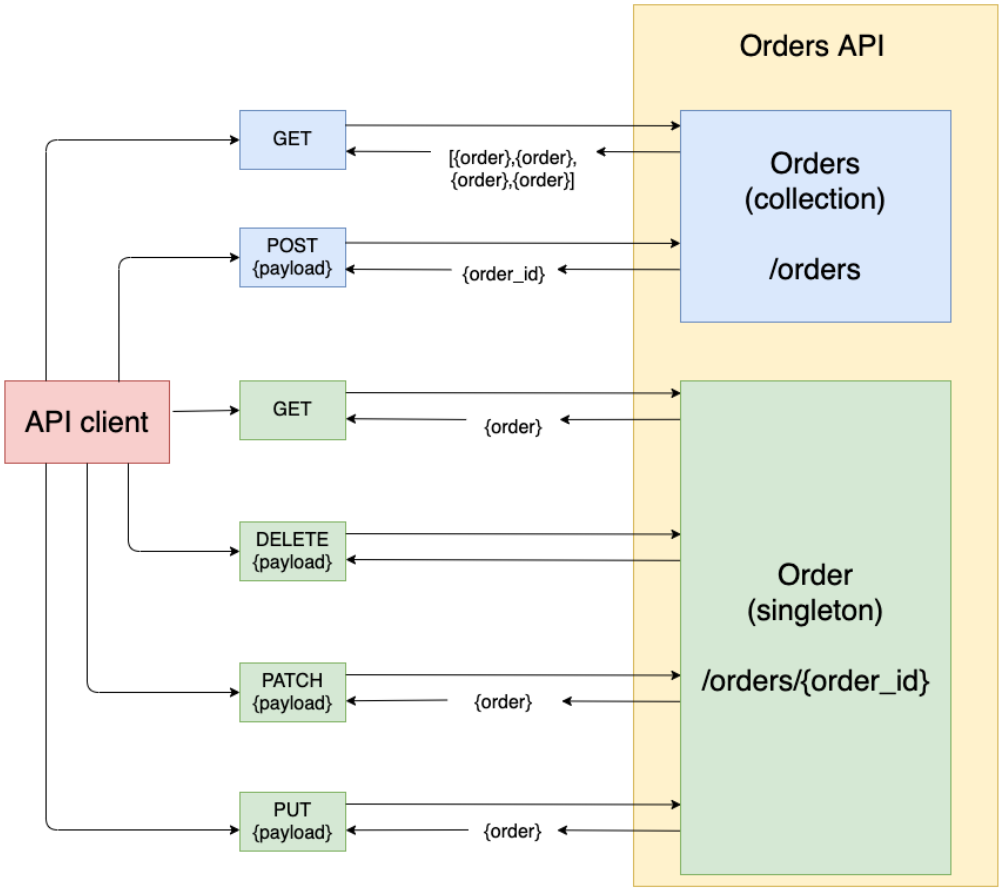


Figure 3.7 In REST, the endpoints of an API are organized around resources. We distinguish two types of resources: collections and singletons. In this illustration, the /orders URL path represents a collection of orders, while the /orders/{order_id} URL path represents a single order. Depending on the URL path and the type of resource it represents, we use different HTTP methods to perform actions on the resource.

RESTful APIs are usually documented using the Swagger/OpenAPI standard. This standard was originally created in 2010 by Tony Tam under the name of Swagger API. The project gained in popularity, and in 2015 the OpenAPI Initiative was created to maintain the specification. In 2016, the specification was officially released under the name of OpenAPI Specification (OAS). As of the time of this writing, the latest major release of OAS is version 3. We will discuss this specification in more detail in chapter 5, but suffice it to say that, in order to document your APIs with the OpenAPI specification, you typically create a YAML or a JSON file where you use a number of fields to describe the endpoints of your API and the data schemas which are accepted in the endpoints. YAML is a data serialization format commonly used for configuration files and documentation. YAML's syntax is a subset of JSON, and for most common use cases, YAML files are fairly portable to JSON and vice versa. The name YAML is a recursive acronym for YAML Ain't Markup Language.

The data exchanged through a REST API goes in the payload body of an HTTP request/response. This data can be encoded in any type of format that the producer of the API wishes to enforce, but it is common practice to use JSON.

Thanks to the possibility of creating API documentation with a high level of detail in a standard specification format, and thanks to the strong emphasis that the REST paradigm makes on the separation of concerns between API layer and application logic, it makes an ideal candidate for enterprise API integrations, and for the creation of APIs with a large and diverse range of consumers.

3.1.5 Web APIs in GraphQL

This section explains what GraphQL is and how it compares to REST. GraphQL is a query language based on notion of graphs and nodes. As of the time of this writing, it is one of the most popular choices for the implementation of web APIs, together with REST. It was developed for internal by Facebook in 2012, and publicly released in 2015. GraphQL is as a protocol-agnostic query language, although it's most often implemented on top of HTTP. GraphQL is designed to address many of the issues that RESTful APIs often present.

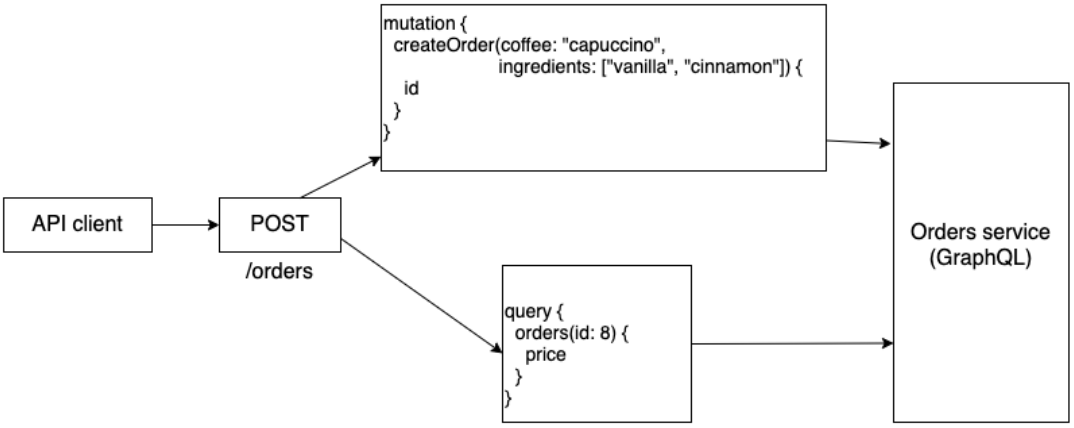


Figure 3.8 Flow of requests in a GraphQL API. Contrary to REST, in GraphQL all requests are directed towards the same endpoint (/orders), and we can use the same HTTP method for all operations. In this case, we are using POST with JSON payloads.

One of such issues is the difficulty of representing all possible transactions through pure HTTP elements. For example, let's say you ordered a cup of coffee through the CoffeeMesh website, and shortly after you change your mind and decide to cancel the order. Which HTTP method is most appropriate to represent this action? You can argue that cancelling an order is akin to deleting it, so you could use the DELETE method. But, is cancelling really the same as deleting? Are you going to delete the order from your records after cancellation? Probably not. You could argue that it should be a PUT, since you are actually changing the state of the order to cancelled. Or you could say it should be a POST, since the user is triggering an operation which involves much more than simply updating a record. However you look at it, HTTP does present some limitations when it comes to modeling user actions, and GraphQL gets around this problem by not constraining itself to using exclusively elements of the HTTP protocol.

Another limitation of REST is the inability for clients to make granular requests of data. For example, imagine that you just placed an order through the CoffeeMesh website, and that the identifier for that order is the number 8. When you make a GET request on the `/orders/8` endpoint, you get all the details about the order that you just made. If the endpoint is designed to return a large number of details, the response will include a large payload, even if you are interested in only one or two details, for example its status. This limitation poses serious problems for small devices, such as mobile phones, which may not be able to handle and store large amounts of data, and may have more limited network access.

Similarly, accessing specific resources usually takes a few rounds of requests. For example, how do we know that the last order we made is represented by the endpoint `/orders/8`? We first have to query the `/orders` endpoint with URL query parameters. For example, we may be able to retrieve the last 10 orders made by a specific user by querying the endpoint with the following parameters: `/orders?user_id=3&limit=10`. Out of the list of orders returned by this endpoint, we know that the order with the most recent timestamp is the last order, so we can use its ID to build the URI for that specific order. Before we do this, however, how do we know that our user ID is the number 3? This information might be available on login, and then it's the responsibility of the client application to store this ID somewhere for later use in other requests. Or we may have another endpoint available to retrieve this information. The bottom line is: getting specific details about a specific resource often requires a few rounds of requests, which again, in the case of small devices might not be convenient or even feasible.

GraphQL avoids these problems by allowing clients to make very granular queries to the server, indicating which specific pieces of data should be returned. For this reason, GraphQL makes an ideal candidate for APIs which have to be consumed by clients with limited network access or limited storage capabilities.

3.2 Defining the application boundaries

The first step in the design of the microservices architecture for an API-driven website is to break down the platform into microservices. The process of breaking down a system into microservices is called **service decomposition**. In order to decompose a system into services, we have to define the boundaries between services. We can use different strategies to identify the boundaries that define our services. In this section, we will discuss the following decomposition strategies:

- Decomposition by business capability
- Decomposition by subdomains

In this section, we see how these methods work and we apply them to decompose the CoffeeMesh application into a collection of microservices. As part of this process, we will consider what capabilities the services should expose through their interfaces, and the API endpoints that would correspond to them. In addition to these strategies, the following principles should guide our design of our microservices architecture:

- Database per service pattern
- Loose coupling
- Single Responsibility Principle (SRP)

The database per service pattern is one of the core design patterns in microservices architecture. According to this pattern, each microservice owns a specific set of the data, and no other service should have access to such data except through an API interface. In spite of the name of this pattern, it does not mean that each microservice should be connected to a completely different database. It could be different tables in a SQL database, or different collections in a noSQL database. The main point of this pattern is to ensure that the table or database owned by a specific microservice is not accessed by another microservice.

Loose coupling and the Single Responsibility Principle (SRP) are principles drawn from the field of software development which can provide helpful guidance in the design of our microservices architecture. Loose coupling emphasizes the idea of building independent components with a clear separation of concerns. Loosely coupled components are components which don't rely on the implementation details of each other. When decomposing microservices by following the methods that we discuss in this section, we may find that some components are highly dependent on each other, either because they use very similar datasets, or because they share common logic. In such situations, it's preferable to merge those components within the same microservice.

The Single Responsibility Principle emphasizes the design of components with few responsibilities, and ideally with only one responsibility. When applied to the design of microservices architecture, this means that we should strive for the design of services with one clearly defined responsibility.

3.2.1 Decomposition by business capability

When using decomposition by business capability, we look into the activities that a business performs, and how the business organizes itself to undertake them. We then design microservices that mirror the organizational structure of the business. For example, if the business has a customer management team, we will build a customer management service; if the business has a claims management team, we will build a claims management service; for a kitchen team, we will build the corresponding kitchen service; and so on. For businesses which are structured around products, we may have a microservice per product. For example, a company that makes pet food may have a team dedicated to dog food, another team dedicated to cat food, another team dedicated to turtles, and so on. In this scenario, we will build microservices for each of these teams.

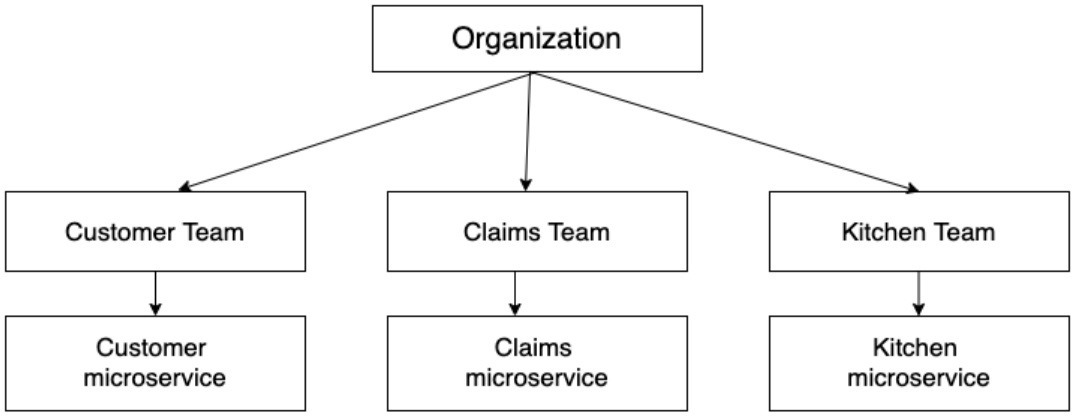


Figure 3.9 Using service decomposition by business capability, we reflect the structure of the business in the architecture of our microservices, therefore we build one microservice for every team or product in the business: for a customer team, we build the equivalent customer microservice, for a claims team, we build the equivalent claims microservice, and so on.

How would this work in the case of CoffeeMesh? Through the CoffeeMesh website, users can order different types of coffee-related products out of a catalogue which is maintained by the **Coffee Team**. This team is in charge of creating new products for the CoffeeMesh website. The team actively monitors user reviews on the existing coffee products, and makes sure that products with bad reviews are improved or removed from the catalogue. The availability of products and ingredients depends on the inventory stock at the time of the order, which is looked after by the **Inventory Team**.

An **Orders Team** is dedicated to improving the experience of ordering products through the CoffeeMesh website. Their goal is to maximize the value of sales through the website, and ensure that customers are happy with their experience and wish to come back. A **Finance Team** makes sure that the company is profitable, and looks after the infrastructure required so that users can make payments or get their money back when they cancel an order.

Once a user places an order, the details of the order are passed on to the kitchen so that production can commence. This process is fully automated, and a dedicated team of engineers and chefs monitors the flow to ensure that no faults happen during production (the **Chef Team**). When the coffee is ready for delivery, it's picked up by a drone which flies the cup of coffee and any other products ordered by the user to the right location. A dedicated team of engineers monitors this process to ensure the operational excellence of the delivery process (the **Delivery Team**).

Out of this structural organization, it seems apparent that we can map the following business teams to microservices:

- **Coffee Team -> coffee service**: this service offers an interface for users to select the product they want to order. This service owns data about the products on offer, but it needs to check the availability of each product and ingredient by consulting with the inventory service. This service offers its interface through the /coffee endpoint.
- **Inventory Team -> inventory service**: this service owns data about the availability of each product and ingredient, and it offers an interface through which we can check for such availability. Updates to the availability of each product and ingredient are done through the

interface provided by this service. This service offers its interface through the /inventory endpoint.

- **Orders Team -> orders service**: this service is the interface of a user to ordering products in the website, and orchestrates the ordering process for the user. It serves as an API Gateway which hides the complexity of the platform for the user, so that the latter doesn't have to know about all different endpoints and what to do with them. It also owns data about orders and their status. This service offers its interface through the /orders endpoint.
- **Finance Team -> finance service**: this service offers a payments interface through which users can pay for their orders. This service owns data about user payment details and payment history, and it offers its interface through the /finance endpoint.
- **Chef Team -> chef service**: this service sends orders to the kitchen with the details of the orders made by users, and it keeps track of the progress in the order production process. It owns data about the orders passed on to the kitchen, and it offers its interface through the /chef endpoint which allows us to send orders to the kitchen and to check the status of the order.
- **Delivery Team -> delivery service**: this service arranges the delivery of the order for the user by a drone once it has been produced by the kitchen. This service knows how to translate the user location into coordinates, and how to calculate the best route to their destination. It also provides an interface to follow the itinerary of the drone, and it owns data about every delivery made by CoffeeMesh. This service offers its interface through the /delivery endpoint.

API Gateway Pattern

The API Gateway pattern is a microservices design pattern where we create a single-entry point for all of our APIs. This single point of entry becomes the interface of our clients to our platform, and helps us to hide and encapsulate the complexity of our APIs and services from the clients. This is especially useful when our microservices expose different types of APIs, such as REST, GraphQL, and gRPC. By placing an API Gateway in front of those services, the client doesn't have to handle different protocols in order to interact with our APIs, since the API Gateway acts as a bridge with a unified interface for the client.

For more information on this pattern, see Chris Richardson, *Microservices Patterns* (Manning, 2018, p. 259-291 https://livebook.manning.com/book/microservices-patterns/chapter-8/point-8620-53-297-0).

Is this division into microservices in agreement with the principles of one database per service and loose coupling? From the description, it appears that every service owns its own set of the data, and communication across services happens through API interfaces. However we do have some coupling between the coffee service and the inventory service. For every product that the coffee service returns to the user, the availability of each of them has to be checked against the inventory service. In this regard, there is an argument here for merging both services together under the same endpoint. Although the inventory service has responsibilities that go beyond the needs of the coffee service, we can say that knowing the availability of each product and ingredient is a necessary precondition for the coffee service to be able to serve such list to the user, and being able to retrieve such information directly from the database would improve the performance of the service. Under these circumstances, it appears to make sense to have only one service to manage the catalogue of products and ingredients and to manage their availability. The final layout of the

microservices architecture of the CoffeeMesh website according to the decomposition by business capability method is shown in Figure 3.10.

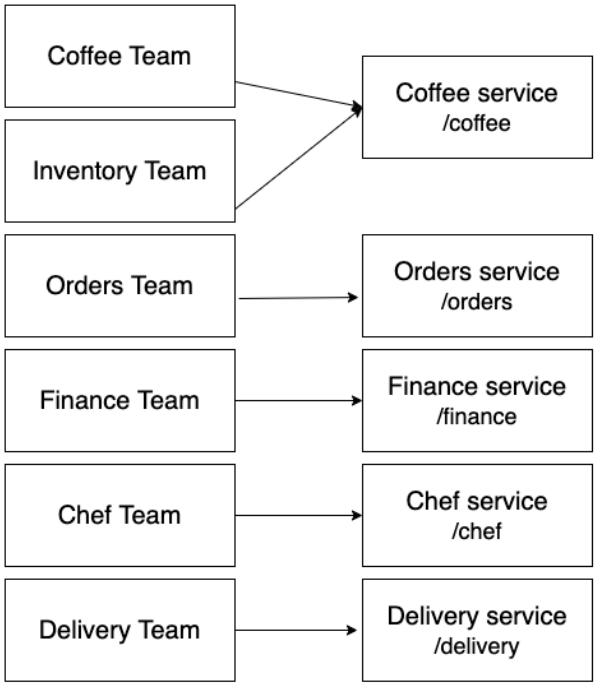


Figure 3.10 Business decomposition applied to the design of the microservices architecture for the CoffeeMesh website: for every team in the CoffeeMesh business organization, we build the corresponding microservice. Each microservice exposes its own API, with their endpoints reflecting the name of the service. In the case of the Coffee Team and the Inventory Team, we found that the data that both teams own is closely related from a technical point of view, and therefore it makes sense to include both capabilities within the same microservice.

In this microservices architecture, we have named every service and every endpoint exposed by such service directly after the business structure that they represent. We have done this for convenience in this example, but it does not have to be that way. For example, the finance service and the /finance endpoint could be renamed to payments service and /payments endpoint, since most user interactions with this service will be happen when users want to make a payment. Similarly, the chef service and the /chef endpoint could be renamed to kitchen service and /kitchen endpoint, since what this service truly provides is an interface to the kitchen where the coffee is produced.

The advantage of decomposition by business capability is that the architecture of the platform aligns well with the existing organizational structure of the business. This alignment might facilitate the collaboration between business and technical teams.

The downside of this approach is that the existing organizational structure of the business is not necessarily the most efficient one. As a matter of fact, it could be outdated and reflect rather old business procedures, instead of the new processes and practices of the organization. If that is the

case, the inefficiencies of the business will be mirrored in the system architecture. Therefore, when designing services that mirror the existing structures of a business, we must carefully analyze how such services fit within the overall technical architecture of the system, and decide whether it really makes sense to reproduce each and every element of the business at the technical level.

3.2.2 Decomposition by subdomains

Decomposition by subdomains is an approach that draws inspiration from the field of Domain Driven Design (DDD). DDD is an approach to software development which helps us to break down a system into loosely coupled components with well encapsulated logic and clearly defined dependencies among them. When applied to the design of a microservices architecture, DDD helps us to provide a definition of the core responsibilities of each service and the boundaries that define the relationships among them. I would like to emphasize that the design of a microservices architecture does not necessarily have to mirror the results of our decomposition by subdomains according to DDD. DDD should play a guiding role in the design of our microservices, but it does not have to determine the architecture.

The methods of Domain Driven Design were best described by Eric Evans in his seminal book *Domain-Driven Design* (Addison-Wesley, 2003), otherwise called "the big blue book". DDD offers an approach to software development that tries to reflect as accurately as possible the ideas and the language that businesses, or end users of the software, use to refer to their processes and flows. In order to achieve this alignment with end users, DDD encourages developers to create a rigorous, model-based language that software developers can share with the end users. Such language must not have ambiguous meanings, and it is called by Evans Ubiquitous Language.

In order to create a Ubiquitous Language, we have to identify the core domain of a business, which corresponds with the main type of activity that an organization performs to generate value. For a logistics company, it can be the shipment of products around the world. For an ecommerce company, it can be the sale of products. For a social media platform, it can be feeding a user with relevant content. For a dating app, it can be matching users. In the case of the CoffeeMesh application, the core domain corresponds with the mission of the company to deliver the best coffee in the world to its customers, in the shortest possible amount of time.

The core domain is often not sufficient to cover all areas of activity in a business, so DDD also distinguishes supportive subdomains and generic subdomains. A supportive subdomain represents an area of the business which is not directly related to value generation, but it is fundamental to support it. For a logistics company, it can be providing customer support to the users shipping their products, leasing of equipment, managing partnerships with other businesses, and so on. For an ecommerce company, it can be marketing, customer support, warehousing, and so on.

The core domain gives you a definition of the problem space, which is the problem that you are trying to solve with software. The solution consists of a model, understood here as a system of abstractions which describes the domain and solves the problem. Ideally, there is only one generic model which can provide a solution space for the problem, with a clearly defined ubiquitous language. In practice, however, most problems are complex enough that they require the collaboration of different models, with their own ubiquitous languages. The process of defining such models is called strategic design.

How does this work in practice? How can we apply strategic design in order to decompose CoffeeMesh into subdomains? In order to break down a system into subdomains, it helps to think about the operations that said system has to perform in order to accomplish its goal. In the case of

the CoffeeMesh application, we are interested in modeling the process from the moment the user hits the website, to the moment the coffee is delivered into their hands. What does it take to accomplish this? Let's examine each step one by one (see figure 3.9 for an illustration of these steps).

1. **Step 1**: When the user lands on the website, we firstly need to show a list of products together with their prices. Each product must be marked as available or unavailable, and must also come with an average user rating if reviews are available for that product. The user should be able to filter the list by availability, by price, and by rating. The user should also be able to sort by price (from minimum to highest and from highest to minimum) and by rating (from minimum to highest and from highest to minimum).
2. **Step 2**: The user must be able to select a product and store it in a basket for later payment.
3. **Step 3**: The user must be able to pay for their selection.
4. **Step 4**: Once the user has paid, we must be able to pass on the details of the order to the kitchen.
5. **Step 5**: Kitchen picks up the order details and produces the product ordered by the user.
6. **Step 6**: The user must be able to monitor progress on their order.
7. **Step 7**: Once the user's order is ready for delivery, we must be able to arrange its delivery with a drone.
8. **Step 8**: The user must be able to track the drone's itinerary until their order is delivered.

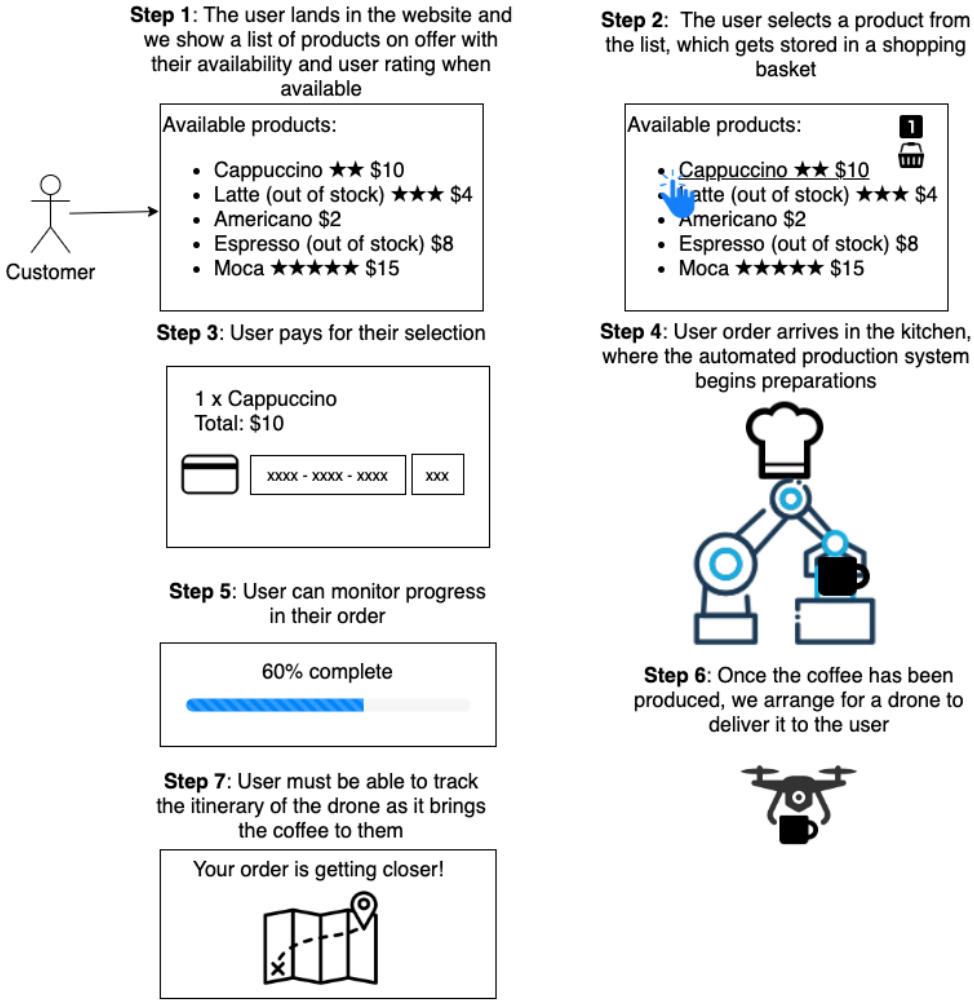


Figure 3.11 Illustration of the user flow from the time they land the website to the time their order is delivered.

The first step requires a subdomain which is capable of providing a list of CoffeeMesh products on offer through an interface. We can call it the **products subdomain**. This subdomain must be able to determine which products are available and which are not. In order to do so, the products subdomain must contain logic capable of tracking the amount of each product and ingredient in stock. Other components of the CoffeeMesh platform can use such interface to decrease the amounts in stock whenever they are used for production.

The second step requires a subdomain which allows users to make a selection of the items returned by the products subdomain and store them in a basket. This subdomain acts as an orchestration framework for the ordering process, and we can call it the **orders subdomain**. This subdomain owns data about the orders made by users, and it exposes an interface which allows us to manage orders and check their status. The orders subdomain serves as an API Gateway which

hides the complexity of the platform for the user, so that the latter doesn't have to know about all different endpoints and what to do with them. The orders subdomain also takes care of the second part of step 4: passing the details of the order to the kitchen once the payment has been successfully processed. It also meets the requirements for step 6, allowing the user to check the state of their order through the interface exposed by this subdomain. As an orders orchestration framework, the orders subdomain also knows how to interface with the delivery subdomain in order to arrange the delivery of the user order, once it has been produced by the kitchen.

The third step requires a subdomain which can handle user payments. We will call it the **payments subdomain**. This domain must provide an interface through which the user can provide their payment details in order to process the payment. The payments subdomain owns data related to user payments, including user payment details, payment status, and so on. It also contains specialized logic for payments processing, including card details validation, communication with third-party payment providers, different methods of payment, and so on.

The fifth step requires a subdomain which knows how to interact with the kitchen to manage the production of the order made by the user. We call it the **kitchen subdomain**. The production system in the kitchen is fully automated, and the kitchen subdomain knows how to interface with the kitchen system in order to commission the production of a user order, and how to track its progress. Once the order is produced, the kitchen subdomain notifies the orders subdomain, which is then able to arrange its delivery. The kitchen service owns data related to the production of the user order, and exposes an interface which allows us to send orders to the kitchen and to keep track of their progress. The orders subdomain interfaces with the kitchen subdomain to update the status of the user order in order to meet the requirements for step 6.

The seventh step requires a subdomain which knows how to interface with the automated delivery system driven by drones. We call it the **delivery subdomain**. This subdomain contains specialized logic to resolve the geolocation of a user based on a given address, and to calculate the most optimal route to reach the user. It also knows how to manage the fleet of drones, and knows how to optimize the arrangement of deliveries. It owns data about the fleet of drones, and also about addresses and their coordinates, as well as data related to all the deliveries made by CoffeeMesh. The delivery subdomain exposes an interface which allows us to arrange a delivery and to keep track of it.

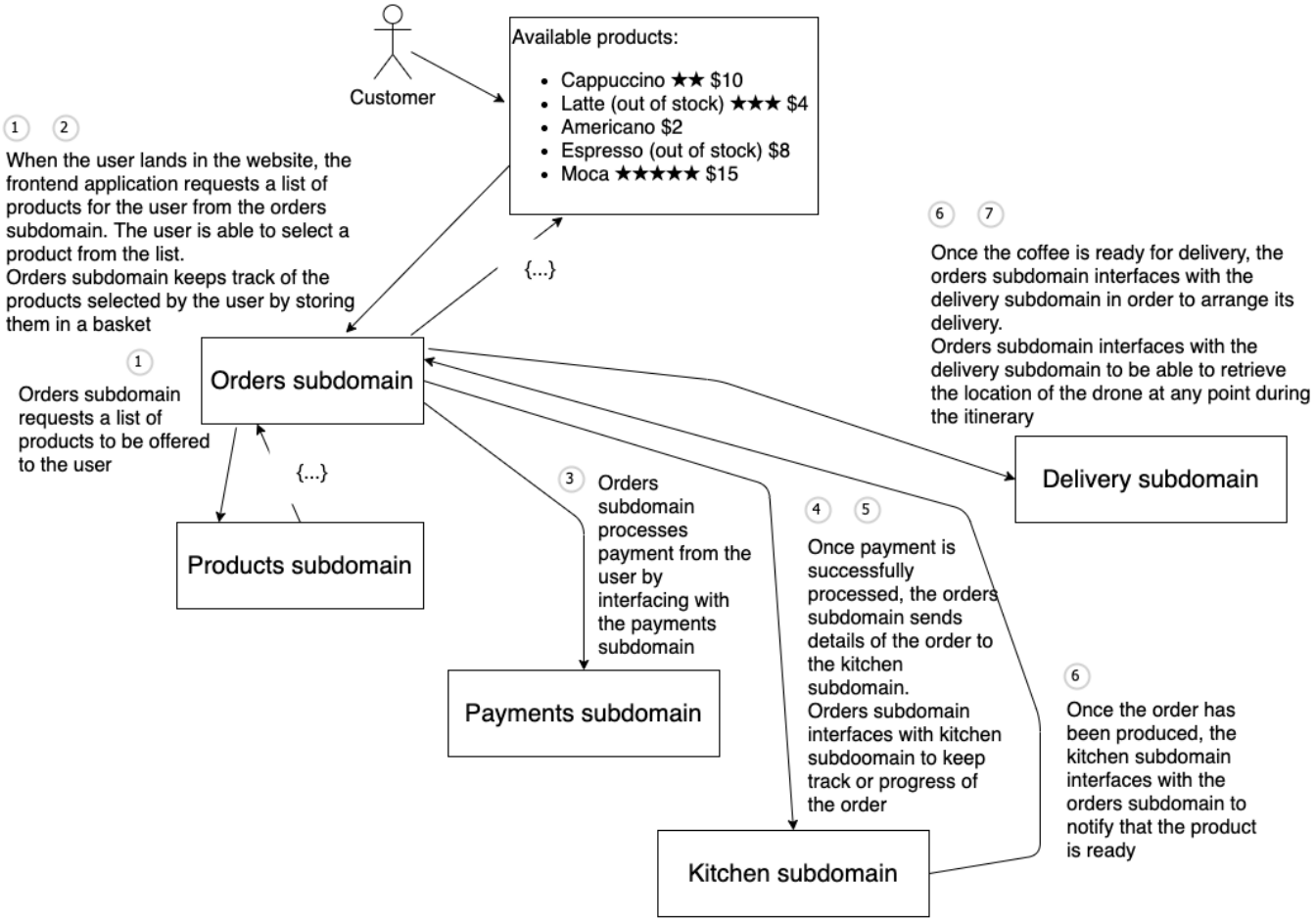


Figure 3.12 Flow of the users from the time the land on the website to the time they receive their orders from CoffeeMesh. Each of the steps in Figure 3.11 is captured by one of the subdomains in this diagram. The text blobs contain references to the flow steps in the encircled numbers around them. The arrows represent the flow of communication: what is talking with or interfacing with what.

Through the strategic analysis of Domain Driven Design, we obtain a decomposition for CoffeeMesh into five different subdomains. These subdomains can be mapped to microservices, as each of them encapsulates a well-defined and clearly differentiated area of logic and owns its own data. None of these subdomains shows strong dependencies towards each other. We can say that these subdomains comply with the principles enumerated at the beginning of this chapter, namely the database per service patter, the principle of loose coupling, and the single responsibility principle. The resulting microservices architecture is the following:

- **Products subdomain -> products service** exposing its interface through the /products endpoint.
- **Orders subdomain -> orders service** exposing its interface through the /orders endpoint.

- **Payments subdomain -> payments service** exposing its interface through the /payments endpoint.
- **Kitchen subdomain -> kitchen service** exposing its interface through the /kitchen endpoint.
- **Delivery subdomain -> delivery service** exposing its interface through the /delivery endpoint.

As it turns out, the results of this analysis are very similar to the results that we obtained from the analysis of decomposition by business capability. Figure 3.13 shows the microservices architecture of CoffeeMesh resulting from the strategic design applied in this section.

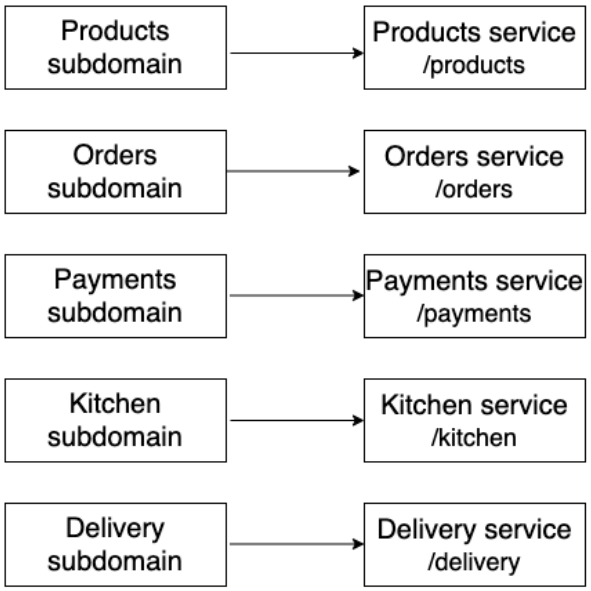


Figure 3.13 The result of applying the strategic design analysis of Domain Driven Design breaks down the CoffeeMesh platform into five subdomains which can be mapped directly to microservices with their own endpoints.

3.2.3 Service decomposition strategies for microservices: practical and organizational matters

Which service decomposition strategy should we use in order to design our microservices architecture: decomposition by business capability, or decomposition by sub-domains? The answer is that, when it comes to microservices architecture, these two strategies are not mutually exclusive, and in practice you will use a combination of both. It helps to decompose the system by business capabilities and by applying the strategic design approach of DDD, and analyze how the results of one approach compare with the other. In most cases, you will find that the two approaches yield very similar results.

As we described in chapter 1 and we will illustrate further in chapter 14, microservices bring additional operational complexity, so when decomposing a system into microservices, make sure you have the necessary resources to manage them. Generally, you should aim for a situation

where each microservice is maintained and operated by a two-pizza team (3 to 12 people). When services are considerably small and simple, it could be the case that a smaller team is able to look after them. And when a collection of microservices is considerably related or dependent on each other, it may make sense that a single team looks after all of them.

If your organization is small, consider combining a monolith application with microservices. If your project hasn't yet grown out of its monolithic architecture, it is easier to maintain and operate a set of applications which are part of the same monolith, than a set of applications which are deployed as microservices. Balance the capabilities of your organization with your system needs, and consider decomposing only critical components which have complex logic and demanding performance requirements into independent microservices, and leave the rest in the monolith. As your organization grows, you can always continue breaking down the monolith into microservices choosing any of the strategies discussed in this section.

3.3 What is API Governance and why should you care?

This section explains the concept of API Governance, what it implies in practice, and how following best API Governance practices in your API design and operations can help you deliver better experiences for your API consumers. As your APIs grow, it may become difficult to keep track of them, and you can lose consistency in your API designs across the stack. Without careful planning, it is not too difficult for an organization to end up in a situation where different teams working on different APIs are trying to do the same things in different ways. For example, they may use different specification standards, different authentication mechanisms, different resource definition strategies, or worse, they may be reinventing and duplicating already existing and well tested code.

A healthy dose of diversity is always welcome and can actually help to drive improvements. However, when a team ends up doing things differently because of lack of coordination with other teams, and most importantly, because of lack of a central strategy for API design and management, there is hardly a benefit to reap.

Organizations which take their APIs seriously, especially those who intend to offer their APIs publicly, as a product, for enterprise integrations, need a centralized strategy to API design and management that can make it easy for their consumers to build and maintain their integrations. API Governance offers a number of best practices which help you achieve this outcome. In this section, we will focus on the following aspects of API Governance:

- API documentation with specification standards
- API versioning
- Managing the lifecycle of your APIs
- API security and authentication

Let's examine how API Governance tackles each of these topics in detail.

3.3.1 API documentation with specification standards

This section discusses the advantages of using a standard Interface Description Language (IDL) to document your APIs, and common pitfalls that can arise when you don't use them. No matter whether you intend to create a GraphQL API, or a RESTful API, or a gRPC API, or any other type of API, the most important thing is that you use a standard interface description language (IDL) to describe the specification for the API. Having an API specified in a standard IDL comes with substantial benefits, among others the fact that, because it's standard, it usually comes with a host

of public libraries and tools that help you create validations, proceed faster and more reliably in your implementation, and improve your productivity.

For example, if you are implementing a RESTful API, using the Swagger/OpenAPI specification, which we will describe in more detail in chapter 3, will allow you to use a linter for your specification, to make sure it's correctly written. You will be able to use libraries which can generate a nicely formatted, and often interactive version of your specification, which you can publish on your website for documentation. You will be able to use libraries that can automatically apply validations at runtime from your API specification. You will be able to automatically generate tests using tools like POSTMAN. And most importantly, you will always be able to run your code against the API specification in your Continuous Integration system, to make sure it never breaks the interface, and therefore the contract which, through your API, you are promising to honor with your customers.

It may sometimes seem that working on a detailed API specification from the beginning is overengineering your approach, especially if you are going to be working on a small API with a few simple endpoints. Furthermore, following a strict specification standard may feel constraining and limiting of your possibilities. I should hasten to say that, when properly used, standard IDLs do not impose such constrains and are definitely not an overkill. More often than not, I have seen teams being slowed down by the lack of a standard API specification, even with the simplest endpoints.

Simply put, lack of specifications creates lack of enforcement, and opens the door for different interpretations of what an API is supposed to do by the consumer and the producer. Also, your first API specification does not need to be perfect from the beginning, and there is no need to spend days or weeks working on it before you jump into the code. On the contrary, specifications can and should evolve over time. A flexible and changing API is not incompatible with well-specified API, and the existence of such specifications will ensure that both consumers and producers of the API always abide by its contract.

3.3.2 Managing API changes with versioning

This section discusses different strategies that you can use to version your APIs and the practical implications of doing so. APIs are very rarely static. As your product evolves, you need to expose new capabilities and features through your API, and this means that you will need to create new endpoints, or make changes to your schema in order to introduce new entities or fields. It is not unusual to see organizations doing these changes without caring to produce a new version of the API. This is bad practice and can lead to unexpected behaviors.

Often, the changes to an API are backwards incompatible, which means clients who are unaware of the new changes will suddenly get failed responses in response to their requests. Part of managing an API is making sure that any changes you make to it do not break the integration that may already exist with other applications, and API versioning serves that purpose.

Two major types of versioning systems are useful for APIs:

- **Semantic versioning (semver)**: this is the most common type of versioning, and it is widely used to manage software releases. It has the following format: MAJOR.MINOR.PATCH, for example: 1.1.0. The first number indicates the major version of the release; the second number indicates the minor version; and the third number indicates the patch version. The major version changes whenever you make a breaking change to your API, for example when a new field is required, which before was not required or did not even exist. Minor versions represent non-breaking changes to the API, such as the

introduction of a new optional field or a new optional query parameter. Consumers of your API expect to be able to keep calling your endpoints in the same way on different minor versions and continue to obtain responses. Patch versions indicate bug fixes. There are more options that can be included in semantic versioning, for example to indicate whether a specific version is a pre-release. For a full description of the specification of semantic versioning, please refer to https://semver.org/

- **Calendar versioning (calver)**: calendar versioning uses calendar dates to version releases. This is especially useful when your APIs change very often, or when your releases are time-sensitive. An increasing number of software products use calendar versioning to manage their releases, including Ubuntu (https://ubuntu.com/), the Python library Twisted (https://github.com/twisted/twisted), the Python dependency management tool `pip` (https://pip.pypa.io/en/stable/news/), and the Python IDE Pycharm (https://www.jetbrains.com/pycharm/download), among others. Amazon Web Services also uses calendar versioning in many of its products, for example in CloudFormation (https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/format-version-structure.html)

  or in the S3 API (https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html). Calver does not provide a full specification about how to format your versions, it just emphasizes the use of dates for your versioning. Some projects use the format YYYY.MM.DD, while some others use YY.MM, and some others, like `pip`, use YY.MINOR.PATCH, where MINOR and PATCH are used in the same sense as in `semver`. If you make several releases per day, you can use an additional counter after days to keep track of each release, for example 2020.04.12.3, which means this is the third release done on the 12th of April of 2020. For more details on calendar versioning, see https://calver.org/ and https://sedimental.org/designing_a_version.html.

You can use any of the above systems to version your APIs, but how do you manage such versioning? You can use different strategies to indicate the version of your API. Which strategy is more convenient can also depend on the versioning system that you use:

- **Versioning using the URL**: you can embed the API version in the URL, for example: `https://coffeemesh.com/api/v1.1.0/coffee`. This is very convenient because consumers of your API know that they will always be able to call the same endpoint and get the same results. If you release a new version of your API, that version will go into a different URL path (`/api/v1.2.0`) and therefore will not conflict with your previous releases. It also makes your API easier to explore, since in order to discover and test different versions of it, API consumers only need to change the version field in the URL. On the downside, when working with RESTful APIs, using the URL to manage versions is considered a violation of the principles of REST, where every resource should be represented by one and only one endpoint.
- **Versioning using the Accept Header field**: the `Accept` HTTP request header field is used by an API consumer to advertise the content types which they are able to parse and understand in a request. In the context of APIs, the typical value of the `Accept` Header is `application/json`, which means the client only accepts data in JSON format. Some proponents argue that you can leverage the `Accept` Header field to also specify the API version that you intend to use. You have to decide the specific format that it can take, but it

can look something like this: `Accept: application/json;v1.1.0`. This approach is sometimes seen as more harmonious with the principles of REST, since it does not modify the resource endpoints.

- **Versioning using custom Request Header fields**: in this approach, you use a custom request Header field such as `Accept-version` and use it to specify the version of the API that you intend to use. This approach is the least preferred, since the use of non-standard elements in the request Headers can lead to integration issues with your clients.

For enterprise solutions, URL versioning is the preferred approach. For one, it offers a very clear and easily browsable interface to users who want to explore different versions of your API. It also offers a more familiar interface to handle errors made by users when specifying the API version that they want to use. Imagine your URL versioning format is `/api/v1.2.0`, and a user types `/api/1.2.0` by mistake. In this case, you can respond with a standard 404 response saying that the resource wasn't found. You can even make sure that all request which do not comply with the right API version format are routed to a specific handler which will inform the user that the API version used is wrong or does not exist. If you are working with a web framework such as Flask or Django, handling requests with the wrong version format is very easy, since you only need to make sure that you capture the right format within application URL routes, and redirect everything else to a specific handler.

If instead you use request Headers to manage API versioning, you are pretty much on your own to build the right functionality that handles errors and typos introduced by the user in the API version format.

3.3.3 Managing the lifecycle of your APIs

This section presents some strategies that you can use to gracefully deprecate your APIs. APIs don't last forever. As the products and services that you offer through APIs evolve and change, some of your APIs will become deprecated and you will eventually retire them. However, your APIs may have external consumers who expect them to be there forever, so you cannot just take them down whenever you please without causing serious disruptions to your clients. You have to manage this process, and as we describe here, you can use specific resources that the HTTP protocol recommends to make the transition explicit and transparent for the consumers of the API.

As soon as you decide that your API is going to be deprecated in the near future, you should announce this to your API consumers through whichever communication channels you have agreed to use with them. For example, you can indicate it in the API documentation. At the same time, you should set the `Deprecation` Header field in your responses. The `Deprecation` Header field is described in an Internet Draft written by Sanjay Dalal and Erik Wilde under the title "The Deprecation HTTP Header Field", which is under discussion for the Standards Track of RFC as of the time of this writing (https://datatracker.ietf.org/doc/draft-dalal-deprecation-header/). If the deprecation date is known, the `Deprecation` field should be set to that date, otherwise it should be set to `true`. For example, if the date is known:

```
Deprecation: Friday, 22nd May 2025 23:59:59 GMT
```

And if the date isn't known:

```
Deprecation: true
```

In addition to the `Deprecation` field, you can also use the `Link` Header field to provide additional information to the consumers of the API. The value of this link is optional, but you have to specify what it is for using optional parameters. For example, if you are providing a link which contains additional information about the deprecation policy, you can set the `Link` field to the following values:

```
Link: <https://coffeemesh.com/deprecation>; rel=”deprecation”; type=”text/html”
```

In this case, we are telling the user that they can follow the link `https://coffeemesh.com/deprecation` in order to find additional information about the deprecation of the API. Alternatively, you can use the `Link` Header field to provide a URL which replaces or supersedes the current API:

```
Link: <https://coffeemesh.com/v2.0.0/coffee>; rel=”successor-version”
```

In addition to the `Deprecation` Header field, you can also use the `Sunset` field to signal when the current URL will become unresponsive. The `Sunset` Header field is described in “RFC 8594: The Sunset HTTP Header Field” by Erik Wilde, which as of May 2019 holds informational status (https://tools.ietf.org/html/rfc8594). For example:

```
Sunset: Friday, 22nd May 2020 23:59:59 GMT
```

The date of the `Sunset` field must be later or the same as the date given in the `Deprecation` header field. The RFC documents mentioned above don’t make specific recommendations about the type of response and status code that should be sent back to the user after the date set in the `Sunset` field has passed. You may use any combination of 3xx and 4xx status codes. A good option is 410 (Gone). 410 is used to signal that the requested resource no longer exists for a known reason. This is different from a 404 (Not Found) response, which indicates that the requested URI doesn’t exist, but the reasons for that are unknown, or the server doesn’t wish to share that information with the user. 301 (Moved Permanently) might be useful in some circumstances. This status code is used to signal that the requested resource has been assigned a new URI, and clients are expected to redirect the user to the newly provide URI. This may or may not work, depending on the nature of your API, so you should carefully consider the requirements of your API in order to choose the most appropriate response status code for it after its deprecation.

3.3.4 API security and authentication

This section discusses the most important security issues that affect your APIs from the point of view of API Governance, and the strategies that you should take to deal effectively with them. This topic will be covered in more depth in chapters 10 and 11. It is important to secure your API endpoint in order to protect your applications against the risk of attacks. We can group these risks into two main categories:

- Operational reliability
- Data access protection

Let’s discuss each of these issues in detail.

OPERATIONAL RELIABILITY

Lack of proper security measures in your APIs can compromise the reliability of your services. We understand by operational reliability the ability of your APIs to service requests within a reasonable amount of time. What is reasonable depends on your benchmarks and the computational intensity of each query.

Different types of attacks which can compromise the reliability of your services, but perhaps the most common is the Denial of Service Attack (DoS, sometimes also called Distributed Denial of Service Attack or DDoS). A DDoS is a type of attach whose purpose is exhaust the resources of your server (CPU and memory) in order to interrupt its service. Whilst this happens, users cannot access your services.

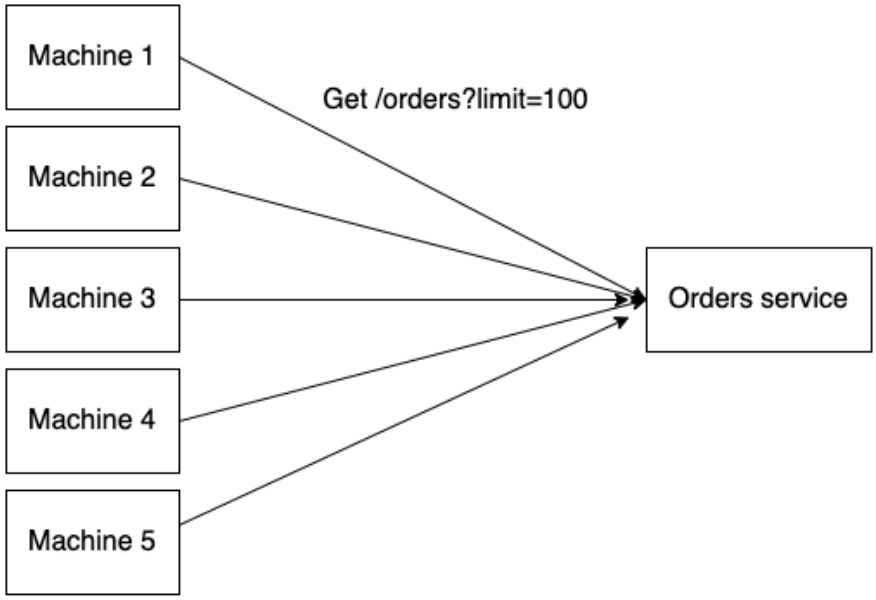


Figure 3.14 In a Distributed Denial of Service (DDoS) attack, an attacker launches multiple requests per second from different machines in order to overwhelm the server. In some cases, these requests are specifically designed to trigger expensive calculations in the server. In this example, the attacker is requesting lists of 100 orders in every request, which eventually will exhaust the resources in the server since the data has to be loaded in memory for every request in order to create a response.

We can distinguish two main categories of DDoS:

- The first type of attack is when the attacker triggers a large number of requests per second until your server is overwhelmed and eventually stops responding to requests.
- The second type of attack is when the attacker discovers certain queries that take a large amount of resources to be serviced, for example queries that require calculations over a large data set which has to be loaded in memory at once. The attacker exploits this vulnerability by launching numerous requests which require this type of calculation, until your servers run out of memory resources, and therefore become unable to continue servicing requests.

In order to guarantee the operational reliability of your APIs, you need to make sure you have measures in place to deal with this type of attacks. We will discuss the characteristics of these and other similar types of attacks in chapter 11, as well as the measures that you can take to protect your APIs against them.

DATA ACCESS PROTECTION

If your endpoints allow user-specific access, most likely you will be managing user accounts with different degrees of user permissions and user-specific data. In many cases, user data can be highly sensitive, and you have an obligation to ensure that it is well protected from unauthorized access. You have to consider different levels of protection and security measures in this area, such as data encryption, securing access to data stores, and others. In this section, we want to emphasize the aspect of security which is directly related to the API layer of your services, namely request authentication and authorization.

Authentication refers to the ability to verify the identity of users trying to access your platform, while authorization refers to the ability to determine whether a specific user has access to a specific resource or operation. It is important that you choose a robust solution to manage API authentication and authorization, and that you make this solution available to all your endpoints. This means that a user will face the same authentication and authorization requirements when they access the `/coffee` endpoint and when they access the `/orders` endpoint, or any other endpoint. You should design your API layer in such a way that you can distinguish different actions and resources, and validate the claims of a user with regards to their ability to access them. We will discuss these topics in more depth in chapters 10 and 11, together with the solutions that you can use to implement your security layers.

3.4 Introducing Documentation-Driven Development

Documentation-driven development is an approach to API development which takes a three-step approach to the implementation of an API:

1. In the first step, you first write the documentation or specification for the API using a standard Interface Description Language (IDL), such as Swagger/OpenAPI for RESTful APIs, or the Schema Definition Language for GraphQL.
2. In the second step, you work on the implementation of your API layer by abiding strictly to the documentation provided in step (1).
3. Validate the implementation against the specification in a Continuous Integration system before it is released. Both consumer/client-side applications and producer/server-side applications must follow this practice.

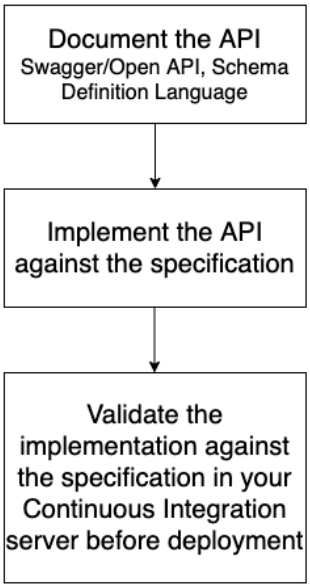


Figure 3.15 The three-step approach of documentation-driven development: document, implement, validate.

This approach is sometimes called API-first method, meaning you first define your API and then you follow the specification in the implementation. However, I prefer the concept of documentation-driven development as it is more specific, and it makes it explicit that you first write the documentation and then use it to guide your implementation. API-first can mean other things as well. For example, it can mean API-first approach to product development, where you design your products by thinking of them first as APIs. It can also mean that you should first implement the interface layer of your server-side applications.

Documention-driven development makes no such assumptions: it only means that, when it comes to implementing the API layer of your applications, you should write the specification first, and then follow it strictly. You can implement application-specific logic before or after that, however you find it more convenient.

Documention-driven development helps you avoid one of the most common problems that API developers face: the often lack of synchronization between consumer-side code and producer-side code. When your API is poorly documented, or documented using non-standard tools which you cannot leverage in your codebase and in your software delivery process, the correct implementation of the API layer in consumer and producer applications relies heavily on the ability of developers to explicitly agree on every detail of the API contract, and in their ability to implement such details with machine precision. Personally, I have never, or very rarely, seen a case in which such approach succeeds its first integration test.

Documention-driven development will give you confidence that, at the very least, consumer- and producer-side code are using the exact same API specification.

In this book, we will show you how to use this approach to make sure that you deliver reliable implementations of your APIs. We will give examples with two types of APIs: RESTful APIs and GraphQL APIs. In both cases, we will use a standard IDL to describe the specification for an API:

Swagger/OpenAPI for REST, and Schema for GraphQL. We will explain the main features of these IDLs and work through examples to see how they are used in practice. We will then explain a number of patterns and strategies that you can use to leverage the provided specifications in order to make your development process easier, more effective, and more reliable.

I would like to finish this section by noting that documention-driven development does not mean that you have to spend days or weeks writing the perfect specification for your APIs, before you start working on the implementation. Your APIs specifications should evolve over time. This means that you can start with a simple and possibly incomplete specification, and evolve it as you know more about the capabilities that you intend to expose through your API. Your APIs will always be changing, regardless of whether you take a documentation-driven approach or not. The advantage of using documentation-driven development is that the changes will be explicitly documented, and you will be able to validate your current implementation against them. This means that you will be able to catch any missing features early in the development cycle, instead of later on, once you have already deployed your API. Combined with API versioning, which we explained in section 3.3.2 of this chapter, this approach will save you a considerable amount of time in your development cycle, and will make your software delivery process more reliable.

3.5 Summary

- We can use different protocols and architectures to implement our web facing APIs. In this chapter, we discussed API protocols:
    - Remote Procedure Control (RPC) allows you to make direct calls to the code running in another machine. It doesn't offer a standard way to document and enforce the API, and therefore it's more suited for internal API integrations.
    - Simple Object Access Protocol (SOAP) is an XML-based protocol for the implementation of web APIs which became very popular in the early 2000s. However, in recent years it has been superseded by newer choices such as REST and GraphQL due to limitations in the protocol.
    - gRPC is an implementation of RPC developed by Google which uses Protobuf for data serialization. It's a highly optimized protocol which can be very useful for private API integrations where performance is a crucial factor.
    - Representational Stater Transfer (REST) is an architectural style which structures its endpoints around the concept of resource and emphasizes stateless communication between client and server. REST APIs are documented using the OpenAPI standard. REST is one of the best choices for enterprise integrations with a large and diverse base of API consumers.
    - GraphQL is a query language which addresses some of the limitations of REST. It allows for more granular requests of data from the server using less transaction cycles, which is very suitable for small devices with lower networking and storage capabilities.
- The process of breaking down a system into microservices is called service decomposition. In order to decompose our system into services, we can use the following strategiers:
    - Decomposition by business capability reflects the organizational structure of the business into the microservices architecture. This helps to align the business with the design of

our architecture, but it can also reproduce the inefficiencies of the business organization into the platform architecture
    - Decomposition by subdomains applies Domain Driven Design in order to model the processes and flows of our business through subdomains. We use the results of this analysis to map microservices to specific subdomains or groups of subdomains.
- API Governance defines best practices for the operation of our APIs. In this chapter, we covered the following best practices:
    - We must document our APIs using a standard Interface Description Language (IDL) to help our customers understand our API and interact with it in a reliable manner.
    - Every change to our APIs must be handled with the release of a new version of our API. This helps to ensure that new changes to our API don't break the integrations with existing consumers of the API.
    - When we are approaching the end of the life of our APIs, we must use the `Deprecation` and `Sunset` Header fields to inform our API consumers when we will stop supporting our APIs, so they can prepare for such event.
    - We must protect our APIs and offer our consumers consistent authentication and authorization requirements across our endpoints.
- Documention-driven development emphasizes the idea of writing the API documentation first, so that we can leverage it in our software development process in order to validate our implementation before deployment and allow client and server code to be developed in parallel.

4

Designing a RESTful API

This chapter covers

- The constraints that underpin the design principles of REST APIs
- How the Richardson Maturity Model can help us understand the advantages of REST best design principles
- The concept of resource in REST APIs
- Designing API endpoints by combining CRUD with the concept of resource
- Using HTTP verbs and HTTP status codes to design meaningful requests and responses for REST APIs
- Using URL query parameters and request payloads in designing our API endpoints

The concept of Representational State Transfer (REST) was developed by Roy Fielding to describe an architectural style for applications that communicate over a network. Originally, the idea of REST defined a number of constrains that applications need to meet in order to be considered RESTful. Over time, with the increasing popularity of REST, more detailed protocols and specifications have been developed that give us well-defined guidelines about how to design our APIs. Today, REST is by far the most popular choice for the implementation of web APIs. In this chapter, we study the protocols and specifications defined in REST, and we see how we can use them to design the specification for the API of the orders microservice of the CoffeeMesh application.

We start by discussing the design principles of RESTful APIs, with special focus on the constraints defined by Roy Fielding when he originally developed the concept. While doing that, we also explain the concept of resource, and what it means for the design of REST APIs. After learning what makes an API RESTful, we study the concept of CRUD. CRUD is an abbreviation for the operations of creating, reading, updating, and deleting. CRUD serves as a notional idea of the operations that a user should be able to perform on resources, and through this chapter we see how it can guide our design of API endpoints.

RESTful web APIs are built on top of the HTTP protocol, and good design practices encourage us to leverage all of the resources that the standard HTTP specification provides to create meaningful communication between a client and a server. In this chapter, we study how to correctly use HTTP verbs and status codes to return meaningful responses to our API consumers. We also see the difference between HTTP payloads and URL query parameters, two important but differentiated concepts which describe different ways in which clients can send data to a server.

4.1 Design principles of REST APIs

This section explains the design principles of Representational State Transfer (REST) APIs. We begin by explaining what the concept of Representational State Transfer means, and how important the concept of resource is in the context of RESTful APIs. After that we discuss six fundamental constraints that make an API RESTful. After discussing the constraints that RESTful APIs must meet, we introduce the concept of Hypermedia as the Engine of Application State (HATEOAS), and discuss what it implies for the design of RESTful APIs. Finally, we introduce the so-called Richardson Maturity Model, which is a mental model developed by Leonard Richardson for describing the degree to which web APIs comply with the constraints of REST discussed in this chapter.

Figure 4.1 shows a diagram flow of a REST API, where an API client can perform a number of actions on a specific endpoint in order to interact with a resource. Each action is mapped to a specific HTTP method. On the other side of the interface, a varying number of instances of the web application serve requests for the API client, while a single entry point for API wraps such complexity for the API client. In the following sections, we delve deeper into each of the elements in this illustration.

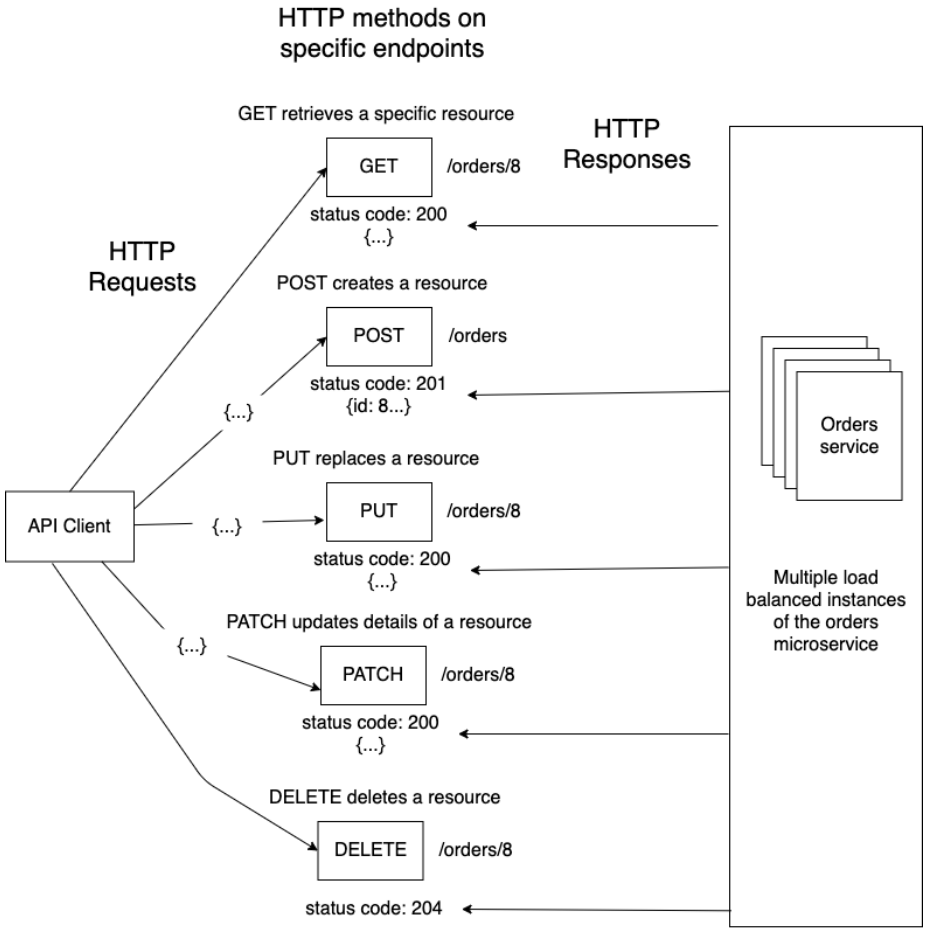


Figure 4.1. Flow diagram of a Representational State Transfer (REST) API. An API client can use different HTTP methods to perform different actions on the /orders API endpoint, such as GET to retrieve details about a specific order, or POST to create an order. On the server side, we can have any number of instances of the same service in order to load balance requests, thanks to the stateless nature of the protocol. The server responds with the expected HTTP status code and a payload. The response for a POST request includes the ID of the resource that has been created.

4.1.1 What is Representational State Transfer?

This section explains what Representational State Transfer means and discusses six fundamental constraints that make an API RESTful. REST stands for Representational State Transfer, a term coined by Roy Fielding in his doctoral dissertation *Architectural Styles and the Design of Network-based Software Architectures* (PhD Dissertation, University of California, Irvine, 2000), to describe an architectural style for loosely coupled and highly

scalable applications that communicate over a network. It must be emphasized that Fielding's original definitions are more focused on application architecture than on API design, and were intended to "evoke an image of how a well-designed Web application behaves"[1]. Modern best practices on RESTful API design have been developed on top of the principles developed by Fielding for the design of RESTful applications.

Representational State Transfer refers to the act of transferring (or communicating) the representation of the state of a resource. The concept of resource is fundamental in the context of RESTful applications. In fact, resources are the key concept around which we design and structure RESTful APIs. An API specification is a collection of endpoints which represent different resources that can be manipulated through the API, together with actions that we can perform on the resources. In REST APIs built on top of HTTP, the actions that we can perform on resources are represented by HTTP methods, such as GET, POST, or PUT (see section 4.3.1 for a more detailed explanation about this).

A resource is an object with a type and associated data that can become the target of a hypertext reference in an API; that is, it's an object which can be pointed to with a specific URL. As we will see when discussing the six constraints of RESTful APIs, the URL pointing to a resource must be unique. There are two types of resources: collections and singletons. For example, in this chapter, we develop the specification for the /orders API of the CoffeeMesh application. This is an API which allows us to make and manage orders from CoffeeMesh. The /orders endpoint represents a collection or list of orders. An endpoint representing a specific order, say /orders/8 for order with ID 8, is a singleton because it represents a unique element from the collection.

Some resources can be nested within another resource. Listing 4.1 shows the payload for an order with several items listed in a nested array.

Listing 4.1 Example of payload with nested resources

```
{
    "status": "active",
    "created": "1588720456",
    "order": [
        {
            "product": "cappuccino",
            "size": "small",
            "quantity": 1
        },
        {
            "product": "croissant",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

In some cases, we want to access only specific details about this order, for example its status. For such cases, we can create a nested resource within the /orders endpoint which targets exclusively the status of an order: /orders/8/status. A GET request on this endpoint

[1] Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures* (PhD Dissertation, University of California, Irvine, 2000), p. 109.

tells us what the status of the order is, without returning all the other details about the order.

Through nested resources it's also possible to represent sub-collections which are related to another resource. Sub-collections are useful in terms of API design as they make it easier to access related resources through the same endpoint. For example, in the CoffeeMesh application, every user gets a user ID, and their profiles are accessible on the `/users` endpoint, so user with ID 1 is represented by the endpoint `/users/1`. In terms of API design, it makes sense to make the orders made by a user available under the /users endpoint, since there's a one-to-many relationship between orders and users: an order can only be bound to a user, but a user can be bound to multiple orders. By making orders a sub-collection of the /users endpoint, we can list the orders made by user with ID 3 under `/users/3/orders`, and we can access the details of a particular order by specifying the ID of that order through this endpoint: `/users/3/orders/8`. Orders are still available through the `/orders` endpoint, but in order to make the API easier to navigate, we can make them accessible under the `/users` endpoint as shown in this paragraph.

4.1.2 Constraints of a RESTful application

In this section, we examine six constraints that Roy Fielding described in his dissertation as defining features of the REST architectural style. These constraints specify how a server should process and respond to a request. Fielding developed the concept of REST in parallel to his contributions to the standardization of the HTTP 1.1 protocol, and originally, he intended to use the concept of REST to describe how web applications should work. Although REST is not necessarily bound to the HTTP protocol, in practice the constraints that we discuss in this section are heavily biased towards web architectures. Let's first provide a brief definition of each constraint before delving into the details:

- Client-server architecture: the user interface must be separate from the server components.
- Statelessness: the server must not manage state with regards to user requests.
- Cacheability: requests that always return the same response must be cacheable.
- Layered system: the server architecture must be structured in layers, but such complexity must be hidden from the user.
- Code on demand: if necessary and if possible, the server should be able to inject code into the user interface on demand.
- Uniform interface: the servers must provide a consistent interface for accessing and manipulating resources.

Let's discuss each of these constraints in more detail.

SEPARATION OF CONCERNS: THE CLIENT-SERVER ARCHITECTURE PRINCIPLE

REST places foremost importance in the principle of separation of concerns, and in consequence, it requires that user interfaces are decoupled from data storage and server logic. This allows for server-side components to evolve independently from UI elements. A common implementation of client-server architectural pattern is by developing the user

interface as a Single-Page Application (SPA) which is deployed independently from the backend application. Figure 4.2 shows an illustration of this architectural constraint.

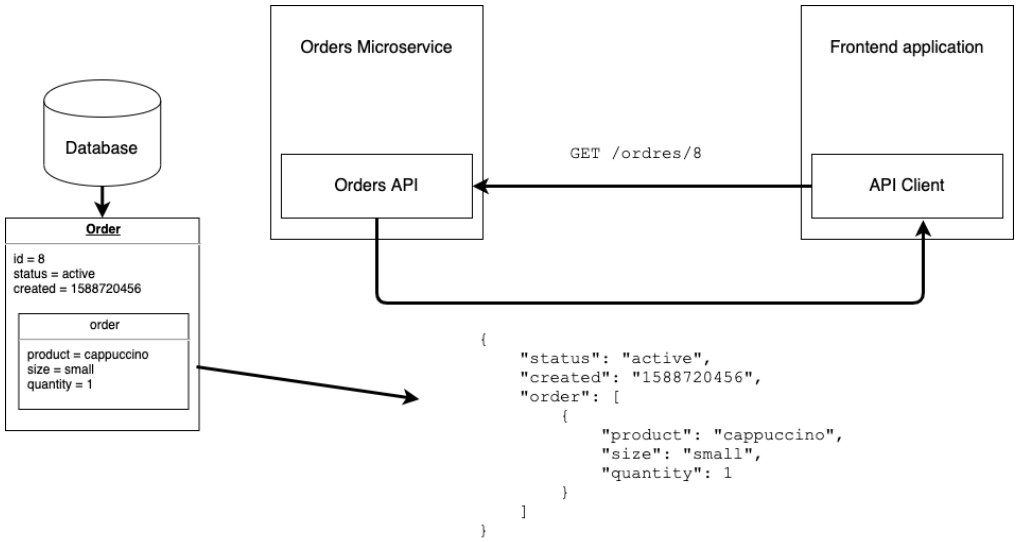


Figure 4.2. Client-server architecture in a REST API. The application that exposes the Orders API, namely the orders microservice, is designed as a component which is independent from the client application, which in this case is represented by a frontend application. The client application contains an API client layer which communicates with the API layer of the orders microservices in order to exchange data. In this illustration, the API client requests details about the order with ID 8 from the server, and the latter responds with a payload which contains a representation of the resource.

MAKE IT SCALABLE: THE STATELESSNESS PRINCIPLE

In REST, every request to the server must contain all the necessary information to process the request. Server-side applications must not keep state from one request to the next. Among other things, this means that the server should not manage user sessions. Removing state management from server components makes it easier to scale them horizontally since they don't need to share state. Figure 4.3 illustrates this concept by showing the Orders microservice deployed with multiple instances running in parallel, and an API client which is able to create an order by sending a POST request to one of the instances, and then check the status of the order with other instances of the service. Because none of the instances are managing state about the API client or about the order, the API client is able to communicate with any of them about the same resource.

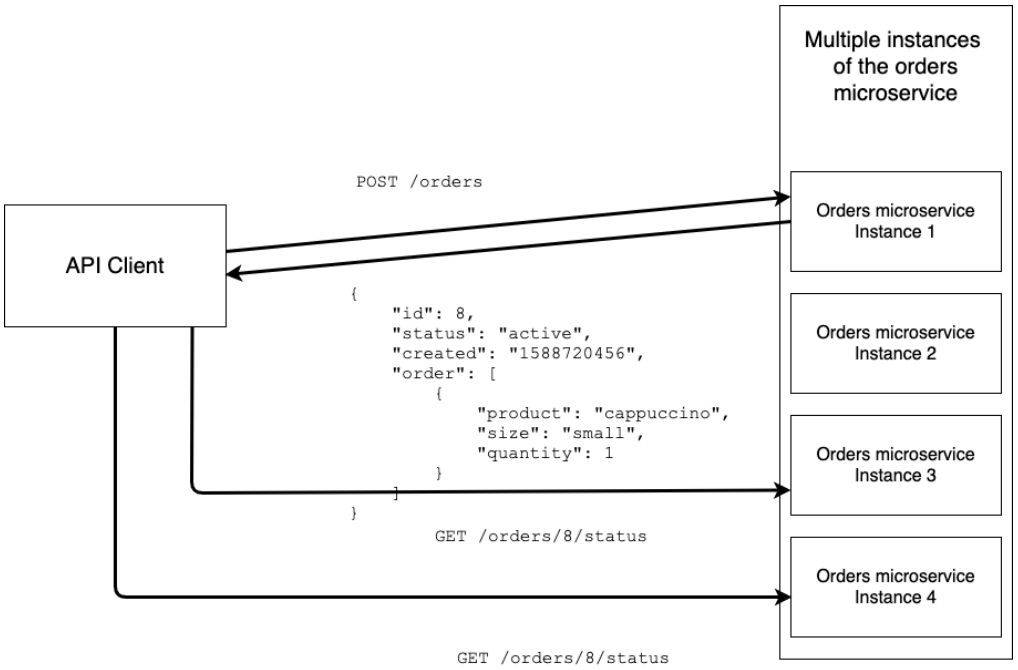


Figure 4.3. The Orders microservice is deployed with multiple instances of it running in parallel. An API client communicates with one of the instances of the service in order to create a, order by sending a POST request, and the same instance sends a response to the client with a payload which contains the ID of the order just created. This information is used by the API client to form requests that check the status of the order just created with other instances of the Orders microservice. Each request sent by the API client contains all the information needed to be processed, and therefore the API client is able to communicate with any instance of the service.

OPTIMIZE PERFORMANCE: THE CACHEABILITY PRINCIPLE

When applicable, resources should be cached in the server. Responses must be explicitly defined as either cacheable or non-cacheable. Caching responses is important for the performance of our APIs, because it means that we don't have to perform again and again all the calculations required to serve a response. We can use any type of caching technology that suits the requirements of our architecture. This principle applies particularly to GET requests aimed at retrieving details about particular resources from the server. Such requests should be cacheable, which means they should not accept request payloads (see section 4.3 within this chapter for more clarifications on this point). For example, a GET request on the /orders endpoint returns a list of identifiers of the orders made by the user who makes the request, as shown in listing 4.2.

Listing 4.2 List of orders returning only the ID of each order

```
{
    "orders": [
        {
            "id": "5517b5aa-9774-4271-bd49-bd91fcce3af6"
        },
        {
            "id": "9754be82-9fe0-4248-ae4c-a3911a23ad2e"
        },
        {
            "id": "5e9346c9-f6d1-4f0f-83e1-547040f1cd0d"
        },
        {
            "id": "26a09fdf-b4a7-4f8c-8085-21fe53cecd64"
        }
    ]
}
```

This list is unlikely to change. If a user cancels an order, the order is not deleted from the system, but its status is updated to "cancelled". Therefore, requests on this endpoint are perfectly cacheable, and the API should be implemented in such a way that we cache these responses in order to optimize the response cycle.

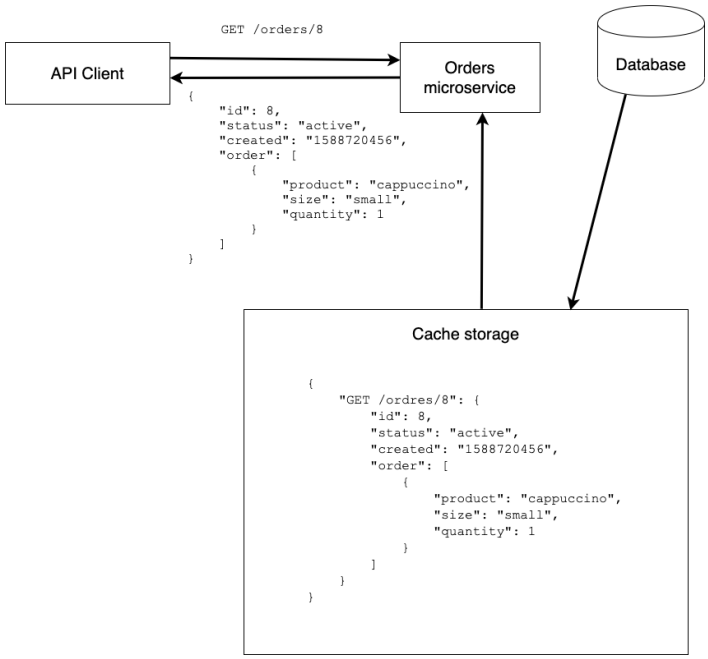


Figure 4.4. The orders microservice uses a cache management storage in order to cache GET requests coming from the client. The allows the orders microservice to serve the requests more efficiently than by having to load the data from the database. In this illustration, when the API makes a GET request on the /orders/8 endpoint, the microservice responds by sending the payload which is already saved in the cache.

MAKE IT RELIABLE: THE LAYERED SYSTEM PRINCIPLE

In a RESTful architecture, a user should have a unique point of entry to your API and should not be able to tell whether they are connected directly to the end server, or to an intermediary layer such as a load balancer. You can deploy different components of a server-side application in different servers, or you can deploy the same component across different servers for redundancy and scalability. This complexity should be hidden from the user by exposing a single endpoint which encapsulates access to your services.

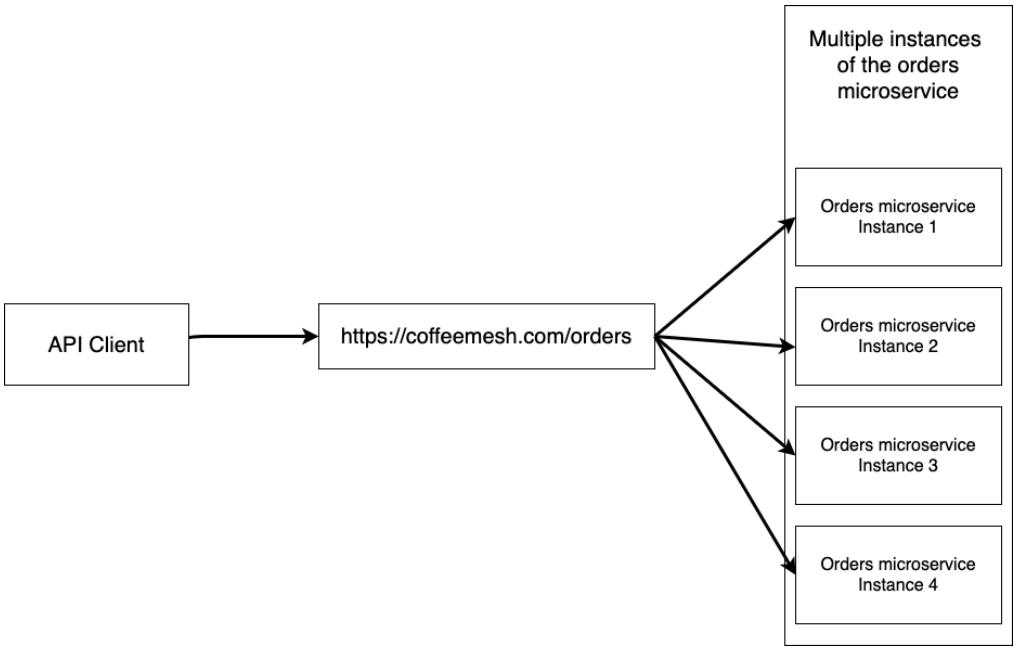


Figure 4.5. In this illustration, we are running multiple instances of the orders microservice in parallel. However, when communicating with the service, the API client is not aware of how many different instances of the service are running at the same time, as it only has to deal with a single-entry point to the service.

EXTENDABLE INTERFACES: THE CODE ON DEMAND PRINCIPLE

Servers can extend the functionality of a client application by injecting executable code directly from the backend. This constraint is completely optional and only applies to architectures which include a user facing application.

KEEPT IT CONSISTENT: THE UNIFORM INTERFACE PRINCIPLE

RESTful applications must expose a uniform and consistent interface to their consumers. The interface must be documented and followed strictly by the server and the client. Also,

individual resources are identified by a Uniform Resource Identifier (URI)[2]. A single Uniform Resource Identifier should always return one and the same resource. For example, for an order with ID 8, the URI for the order is /orders/8, and a GET request on such URI should always return a representation of the state of the order with ID 8. If such order is deleted from the system, the ID must not be reused to represent a different order. Instead, we should return a response with status code 404 (Not Found – see section 4.3.2 for more details on this), informing the user that the requested resource cannot be found in the system.

Resources should be represented using a serialization method of choice, and that approach should be used consistently across the API. Nowadays, RESTful APIs typically use JSON as the serialization format to represent resources, although other formats are also possible, such as XML.

4.1.3 Hypermedia as the Engine of Application State (HATEOAS)

This section discusses the concept of Hypermedia as the Engine of Application State (HATEOAS), which is a paradigm in the design of RESTful APIs which emphasizes the idea of allowing users to discover the capabilities of an API by exposing lists URLs connected with a resource in its representation payload. We'll discuss what this means in practice, and the benefits and disadvantages of this approach. It's worth noting from the start that not all production APIs running on the web follow the guidelines of HATEOAS, or not strictly. Some APIs may use HATEOAS only in certain endpoints which benefit from this approach.

So, what exactly is HATEOAS? In an article written in 2008 under the title "REST APIs must be hypertext-driven" (https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven), Roy Fielding elaborated further on the requirements that an API must meet in order to be considered RESTful. These requirements are relevant for the "Uniform interface" constrain described above. In a nutshell, Fielding suggests that RESTful APIs should include relational links in their response payloads, to allow clients to navigate the API by following such links. The idea is that, departing from a single endpoint, for example a request to retrieve information about a particular resource, a client should be able to navigate the rest of the API based on the links provided in the responses to the client.

[2] For the latest specification on Uniform Resource Identifier, see "RFC 7320: URI Design and Ownership" by M. Nottingham, July 2004 (https://tools.ietf.org/html/rfc7320).

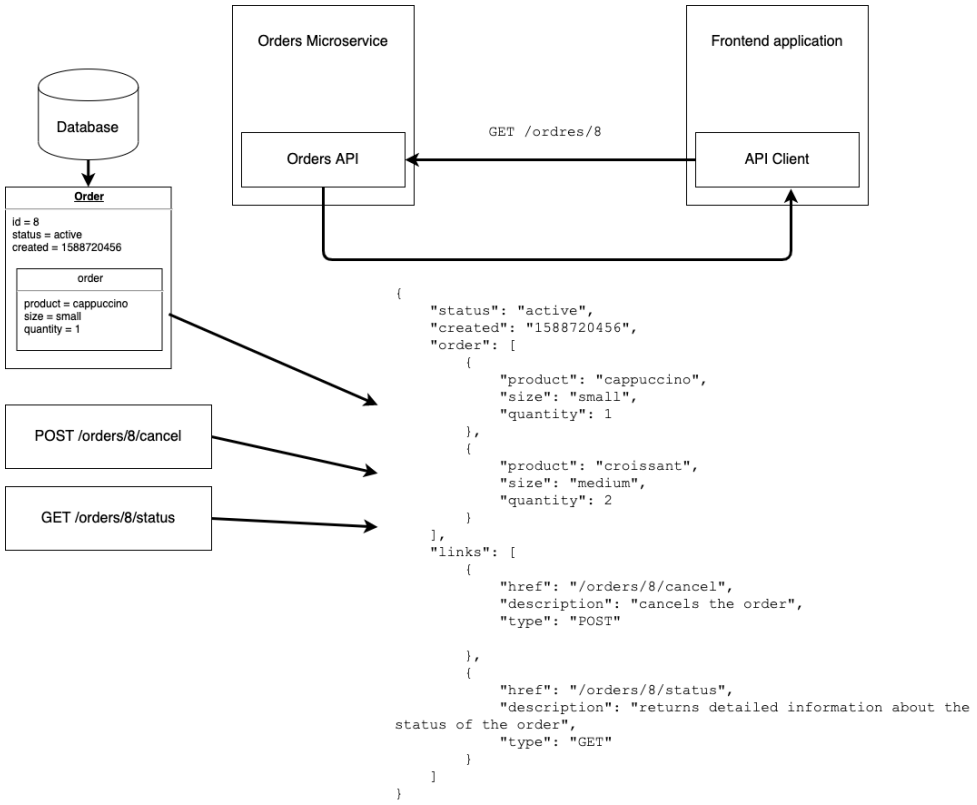


Figure 4.6. Under the Hypermedia as the Engine of Application State (HATEOAS), the API sends to the client a representation of the requested resource, together with links to other resources which are related to the resource. In this case, the API is sending additional endpoints which can be used to perform additional actions on the resource, such as the POST `/orders/8/cancel` endpoint and the GET `/orders/8/status` endpoint.

This idea led to the development of Hypermedia as the Engine of Application State (alias HATEOAS[3]). HATEOAS emphasizes the idea that, when requesting information about a specific resource, a REST client should be able to discover additional resources, endpoints and actions related to that resource. This is accomplished by providing a list of **related links** to the requested resource. For example, if we had made an order through the CoffeeMesh `/orders` API, and we wanted to retrieve the details of that order, the API could provide links to additional endpoints where we can cancel the order, or track its progress, as shown in listing 4.3.

[3] I know, the abbreviation isn't great, and I couldn't tell you why it was chosen.

Listing 4.3 Representation of an order including hypermedia links

```
{
    "status": "active",
    "created": "1588720456",
    "order": [
        {
            "product": "cappuccino",
            "size": "small",
            "quantity": 1
        },
        {
            "product": "croissant",
            "size": "medium",
            "quantity": 2
        }
    ],
    "links": [
        {
            "href": "/orders/8/cancel",
            "description": "cancels the order",
            "type": "POST"

        },
        {
            "href": "/orders/8/status",
            "description": "returns detailed information about the status of the order",
            "type": "GET"
        }
    ]
}
```

This payload is telling us that there are two endpoints associated with this resource, which allow us to perform additional actions on it. The `POST /orders/1234/cancel` endpoint allows us to cancel the order, while the `GET /orders/1234/status` endpoint allows us to keep track of the order status. This idea is further illustrated in Figure 4.6.

The idea of making APIs navigational by providing relational links in each response, as suggested by HATEOAS, is appealing. However, in practice, many APIs are not implemented that way for several reasons:

- The information supplied by hyperlinks is already available to the API client developer, since RESTful APIs already expose documentation in the form of OpenAPI specifications (see section 4.5 within this chapter for details about such specifications). API client developers can use this information to decide which requests they want to make and for what purpose. In fact, the information contained in an OpenAPI specification is far richer and more structured than what you can provide in a list of related links for specific resources.
- It's not always clear exactly what links should be returned. Different users have different levels of permissions and roles, which allow them to perform different actions and access different resources. For example, the average user is able to use the POST /orders endpoint in the CoffeeMesh API to order coffee, and they are also able to use the GET /orders/{coffeeId}/status endpoint to follow the status of their orders. However, they are not able to use the PUT /orders/{coffeeId}/status endpoint

to update the status of their orders, since this endpoint is restricted to specific services which run within the CoffeeMesh platform. If the point of HATEOAS is to make the API navigational from a single point of entry, it wouldn't make sense to return the PUT /orders/{coffeeId}/status endpoint to the average user, since they are not able to use it. Therefore, it would make sense to return different lists of related links to different users according to their permissions. However, this level of flexibility would introduce additional complexity in our API designs and implementations, and potentially lead to coupling between our authorization layer and our interface layer.

- It's sometimes difficult to define the whole extent and breadth of the resources and actions which are related to a specific resource and should be returned for a given user. In fact, depending on the state of the resource, certain actions and resources may not be available. For example, you can call the POST /orders/1234/cancel endpoint on an order with an "active" status, but not on an order with a "cancelled" status. Also, in the payload with details about our order, we could include a link to our user account, since our profile is a related resource of the order. However, does it make sense to send such a link to the user who created the order? This level of ambiguity makes it hard to define and implement robust interfaces that follow the HATEOAS principles.

When working on your own APIs, you can decide whether to follow the HATEOAS principles or not. There's a certain level of benefit in providing lists of related resources in certain cases. For example, in a wiki application, the linked resources section of a payload can be used to list content related to a specific article, links to the same article on other languages, and links to actions which can be done on the article. A list of actions can be particularly useful if not every resource supports the same types of actions. For example, the actions that can be performed on a resource may depend on the current state of the resource, and the concept of linked resources would be useful to let the user know what specific actions can be performed. Overall, you may want to strike a balance between what your API documentation already provides to the client in a more clear and detailed way, and what you can offer in your response payloads to facilitate the interaction of a client with your API.

4.1.4 The Richardson Maturity Model

This section discusses the Richardson Maturity Model, which is a mental model developed by Leonard Richardson to help us think about the degree to which an API is fully compliant with the principles of REST. This model distinguishes 4 levels (from level 0 to level 3) of "maturity" in an API. Each level introduces additional principles of RESTful API design. Let's discuss each level in detail.

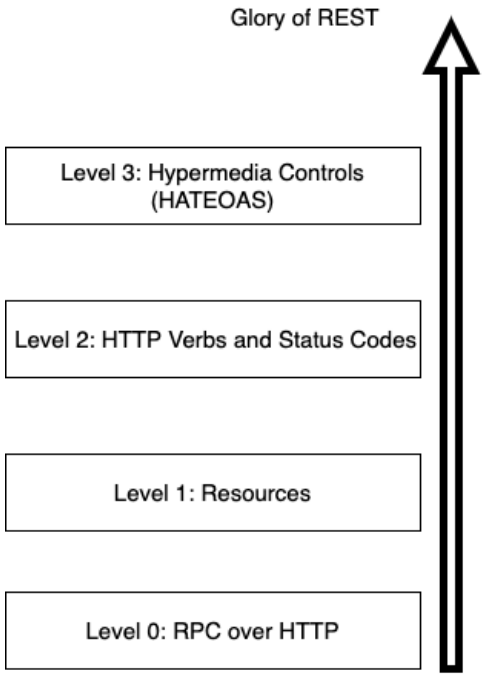


Figure 4.7. The Richardson Maturity Model distinguishes 4 levels of API maturity, where the highest level represents an API design and architecture which is closets to the best practices and standards in REST. Level 0 represents a type of API which relies on HTTP for transport, but doesn't apply any of the concepts of REST, and is therefore closer to an implementation of RPC over HTTP. Level 1 uses the concept of resource representation through URLs in the design of the endpoints. Level 2 takes this step a bit further and introduces the use of HTTP Verbs and Status codes in order to design better endpoints and HTTP responses. Level 3 introduces the concept of linked resources as suggested by the Hypermedia as the Engine of Application State (HATEOAS) paradigm.

LEVEL 0: WEB APIS À LA RPC

In Level 0, HTTP is used basically as a transport system to carry interactions with a server. The notion of API in this case is closer to the idea of Remote Procedure Invocation or RPC (see section 3.1.1 in chapter 3 for more details on this). In this case, all of the requests to the server are done on the same endpoint and with the same (or usually the same) HTTP method, usually GET or POST. The details of what the client is requesting from the server are carried in an HTTP payload. For example, to make an order through the CoffeeMesh website, the client might send a POST request on a generic `/api` endpoint with the following payload:

```
{
    "action": "makeOrder",
    "order": [
        {
            "product": "mocha",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

The server will invariably respond with a 200 response status code and an accompanying payload letting us know whether the request succeeded or not. Similarly, in order to get the details of the most recent order made by a user, a client might make the following POST request on the generic /api endpoint (assuming the ID of the order is number 8):

```
{
    "action": "getOrder",
    "order": [
        {
            "id": 8
        }
    ]
}
```

LEVEL 1: INTRODUCING THE CONCEPT OF RESOURCE

Level 1 introduces the concept of resources. For example, in order to deal with orders in the CoffeeMesh API, a client will interact with an `/orders` endpoint, which serves as a representation of the orders in the system. The request to create an order will be a POST request made on the `/orders` endpoint and would have a similar payload as in Level 0:

```
{
    "action": "makeOrder",
    "order": [
        {
            "product": "mocha",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

However, in order to request the details of the last order made by the user, this time the client will make a POST request on the URI representing that order: `/orders/8`.

LEVEL 2: USING HTTP STATUS CODES

Level 2 introduces the concept of HTTP verbs and status codes. In this level, we use HTTP verbs to represent specific actions that can be performed on resources. In this case, in order to create an order, a client will make a POST request on the `/orders` endpoint with the following payload:

```
{
    "order": [
        {
            "product": "mocha",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

This payload only includes the details of the order placed by the user, and it doesn't include details about the action which is expected from the server, such the "makeOrder" value from the previous levels. In Level 2, the client doesn't need to specify which action is expecting from the server in the payload, because the HTTP verb carries this meaning: in this case, we use the HTTP verb POST to tell the server that we want to create an order. Similarly, in order to get the details of the latest order made by the user, we make a GET request on the URI of such order: `/orders/8`. In this case, we use the HTTP verb GET to tell the server that we want to retrieve details of the resource specified in the URI.

Additionally, Level 2 introduces the concept of HTTP status codes as a way to carry meaningful information to the client. This means that, contrary to the previous levels, every response doesn't invariably get a 200 status code. For example, a POST request gets a 201 response status code, and a request for an inexistent resource gets a 404 response status code. For more information about how HTTP status codes can be used to build meaningful responses, see section 4.3.1 within this chapter.

LEVEL 3: API DISCOVERABILITY

Level 3 introduces the concept of discoverability by applying the principle of Hypermedia as the Engine of Application State (HATEOAS). It does so by introducing the concept of linked resources. In this case, a GET request on the `/orders/8` endpoint in order to retrieve the details of the latest order made by the user would include a list of related links, informing us of the actions that can perform on the resource. This payload is the same as the payload we saw in section 4.1.3 (please refer to that section for an explanation of the payload), which is reproduced here for your convenience as listing 4.4.

Listing 4.4 Representation of an order including hypermedia links

```
{
    "status": "active",
    "created": "1588720456",
    "order": [
        {
            "product": "cappuccino",
            "size": "small",
            "quantity": 1
        },
        {
            "product": "croissant",
            "size": "medium",
            "quantity": 2
        }
    ],
    "links": [
        {
            "href": "/orders/8/cancel",
            "description": "cancels the order",
            "type": "POST"

        },
        {
            "href": "/orders/8/status",
            "description": "returns detailed information about the status of the order",
            "type": "GET"
        }
    ]
}
```

In the Richardson Maturity Model, Level 3 represents the last step towards what he calls the "Glory of REST."

What does the Richardson Maturity Model mean for the design of our APIs? The Richardson Maturity Model gives us a framework to think about where our API designs stand within the overall principles of REST. This model isn't meant to be used in order to grade the degree to which an API design is "compliant" with the principles of REST, or to otherwise assess the quality of an API design. More to the point, the model gives us a framework to think about how we leverage the resources that the HTTP protocol offers in order to create meaningful communication between client and server through a RESTful API.

4.2 Applying CRUD to the design of REST APIs

This section discusses the concept of CRUD and how it can help us guide the design of our APIs. CRUD is an acronym for create, read, update, and delete, and it refers to the four basic operations that a user should be able to perform on data in a persistence storage, such as a database. The concept of CRUD can also be applied in the design of web APIs, and it can help us think about the basic types of operations that a user of the API should be able to perform on the resources represented through the API. Let's see what this means in practice.

In chapter 2, we established that the microservices architecture for the CoffeeMesh website contains a service which services as an orchestration framework for the ordering process: the orders service. The orders service allows the user to place an order, to keep

track of the status of the order, to make changes to it, and to cancel it if they eventually change their mind. These are the capabilities that we want to expose through the service, and for such capabilities we need to design an interface which allows the user to command such actions on the service. How can we represent these capabilities through an API, and how can the concept of CRUD help us design the interface for such purpose? In the following paragraphs, we will see how we can map each of the capabilities of the orders service to a specific CRUD operation and a specific URL paths. This analysis will be extended in section 4.3 of this chapter, when we consider best practices in the use of HTTP methods and status codes to model our request and responses.

A qualification about the use of our vocabulary is in order for the lines that follow. In the following paragraphs, we will refer to URL paths instead of endpoints. Technically, in the context of REST APIs, an endpoint is the combination of a specific URL path and a specific HTTP method. For example, `POST /orders` is specific endpoint of the /orders URL path. More loosely, we can refer to a URL path as an endpoint, and in practice software developers do it all the time. However, for the purposes of the current analysis we need to be precise, and therefore we will refer only to URL paths in this section.

With those clarifications out of the way, let's see how each of the operations represented by the CRUD acronym relates to the actions that a user can perform on their order. **Create** means the ability to create a resource. In the context of the orders API, this means the ability to place an order in our system. Because we don't know what the ID of the order will be before we place it, this operation should be performed directly on the `/orders` URL path.

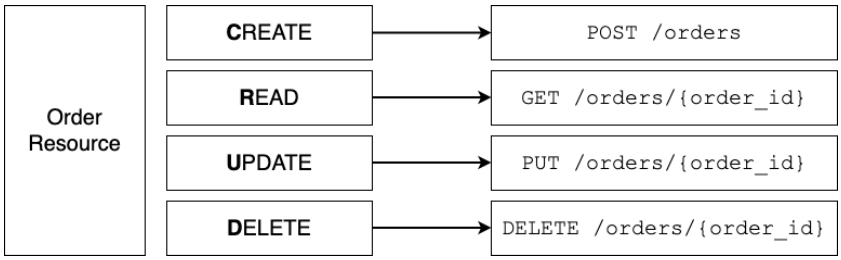


Figure 4.8. CRUD is an abbreviation for the four basic operations that any user should be able to perform on a resource, namely create, read, update, and delete. In a REST API, we should be able to create CRUD endpoints for every resource represented by the API. In this illustration, we are mapping a specific endpoint of the API to each of the CRUD operations that we can perform on an order resource.

Read refers to the ability to retrieve data about a resource, or to be more technically accurate, to obtain a representation of the state of a resource. In the context of the orders API, this means being able to obtain the details of an order placed by the user. In order to obtain the details of a specific order, we need to know its ID, and therefore such request must be done on the `/orders/{order_id}` URL path, where `{order_id}` is a placeholder that must be replaced with the ID of the order whose details we are seeking. `/orders/{order_id}` is a URL which represents the Uniform Resource Identifier (URI) an order, as we explained in section 4.1.2 of this chapter.

Read operations can also be performed on a collection of items. In section 4.1.1 of this chapter, we explained that there are two types of resources: singletons and collections. A singleton is a representation of one particular order, while a collection represents a list of orders. Because a list of orders is not bound to the ID of any particular order, we must be able to obtain a list of orders from the `/order` URL path.

Update is the ability to modify a resource. For example, if our user ordered a small cup of cappuccino, but later changed their mind and decided that they would like to have a big cup of mocha, they would have to update the order. In order to update an order, we need to know the order ID, and therefore we must be able to perform this operation on the URI of the order: `/orders/{order_id}`.

Delete refers to the ability to delete a resource. In the context of the orders API, this means begin able to remove an order from our system. On a first analysis, it might seem like this operation is the equivalent of cancelling an order. I should hasten to say, that's not quite the case! In the modern world of big data, the truth is companies seldom delete their records, unless users specifically request so. Data is a precious commodity, and it can help companies draw insight about market trends and user behavior, among many other things, so normally data is kept in storage and flagged as "deleted". In the context of the orders API, therefore, we can argue that cancelling the order is akin to updating its status from "active" to "cancelled". We should still provide an endpoint which enables to delete an order from our records, but such endpoint will be only for internal use and not exposed to our users. The operation of deleting an order requires knowledge of the ID of the order that we want to delete, and therefore we must be able to perform such operation on the `/orders/{order_id}` URL path.

To summarize the analysis of the previous paragraphs, by applying the concept of CRUD to the design of the orders API, we come up with the following mapping of CRUD operations to specific capabilities and URL paths:

- **Create**: represents the ability to place an order through the `/orders` URL path.
- **Read**: represents the ability to retrieve the details of a specific order through the `/orders/{order_id}` path, or to retrieve a list of orders through the /orders URL path.
- **Update**: represents the ability to update the details of a specific order through the `/orders/{order_id}` URL path.
- **Delete**: represents the ability to delete an order from our records through the `/orders/{order_id}` URL path.

4.3 REST over HTTP: leveraging HTTP for meaningful API communication

This section explains how we can leverage elements of the HTTP protocol in order to create better and more meaningful communication between REST API clients and servers. We will pick up where we left in the analysis of section 4.2 about mapping specific service capabilities to URL paths in our API, and we will see how we can use specific HTTP methods to define unique endpoints to better represent each of the capabilities exposed through the API.

Best REST API design practices encourage us to make proper use of the HTTP protocol in the design of our APIs. As we saw in the Richardson Maturity Model (see section 4.4.4), a more proficient use of the HTTP protocol in our API designs can be associated with a more "mature" API, and it results in a more structured and elegant way to organize the resources represented by our API and the actions that we can perform on them. In this section, we focus on the use of HTTP verbs and HTTP status codes for the orchestration of the communication flow between the client and the server.

The first heading under this section explains how we can use HTTP methods in combination with URL paths in order to create API endpoints that reflect the capabilities that we want to express through the interface. After that, we look at the possible HTTP status codes that we can send in responses to the clients of our API, and discuss best practices around their use. Finally, we put it all together in order to describe the flow of communication between the client and the server according to the specifications that we have been building until this point.

4.1.5 Using HTTP verbs to create meaningful HTTP endpoints

This section discusses best practices in the use of HTTP verbs (also called HTTP methods) to represent actions on the resources represented by our API endpoints. Proper use of HTTP verbs in API designs helps to make them more understandable and predictable for the consumers of the API, which results in a better user experience. In the real world, you will see that different API vendors often make different use of HTTP verbs. In particular, there's often some confusion about the proper use of certain HTTP methods in the context of REST APIs. Let's clear up that confusion from the beginning by studying the specification for each HTTP method. In the context of web APIs, the most relevant HTTP methods for the design of API endpoints are GET, POST, PUT, PATCH, and DELETE. The following list contains a description of the intended use of each of these methods:

- GET: returns information about the requested resource.
- POST: creates a new resource.
- PUT: replaces a resource[4].
- PATCH: updates specific attributes of a resource.
- DELETE: deletes a resource.

Now, how do we use these methods to define endpoints in our API? From the analysis in section 4.2 of this chapter, we have two URL paths with different operations associated with them. On the /orders path, we need to be able to create resources (aka place orders) and to retrieve a list of resources (aka obtain a list of orders). As per the specification for HTTP methods that we have seen, these operations can be mapped nicely to the following endpoints (see figure 4.9 for an illustration of this relationship):

- `POST /orders`: endpoint to place orders.

[4] As per the HTTP specification, PUT can also be indempotent and be used to create a resource if the targeted resource doesn't exist. However, the specification also highlights:
"A service that selects a proper URI on behalf of the client, after receiving a state-changing request, SHOULD be implemented using the POST method rather than PUT". This means that, when the server is in charge of generating the URI of a new resource, new resources must be created using the POST method, and PUT can only be used for updates. See R. Fielding, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231 (June 2014, https://tools.ietf.org/html/rfc7231#section-4.3.4).

- `GET /orders`: endpoint to retrieve a list of orders.

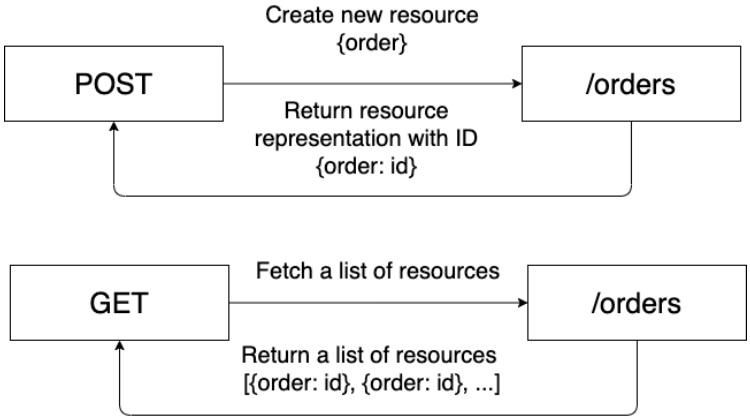


Figure 4.9 Using HTTP method POST to create resources on URL path /orders, and HTTP method GET to retrieve a list of resources from URL path `/orders`

On the `/orders/{order_id}` URL path, we need to be able to perform updates on the order, get the details of the order, and delete the order. You can retrieve the details of the order on this endpoint with a `GET /orders/{order_id}` endpoint; you can delete the order through the `DELETE /orders/{order_id}` endpoint. |Figure 4.10 illustrates this relationship between HTTP methods and actions on the /orders/{order_id} URL path.

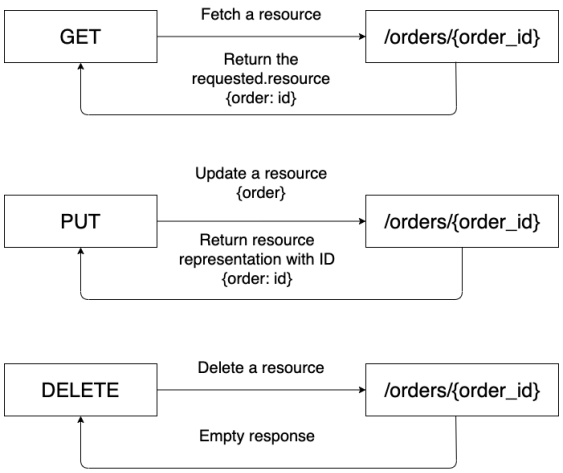


Figure 4.10 Using HTTP method GET to retrieve the details of a resource, PUT to update a resource, and DELETE to delete the resource on URL path `/orders/{order_id}`

What should the endpoint for updating the order look like? As per the specification for HTTP methods, we can choose from two methods in order to define this endpoint: PUT and PATCH. PUT requires the API client to send a whole new representation of the resource, which is then used by the server to replace the existing representation of the resource. In practice, what this means is that the HTTP client has to send all the details of an order, and such details will be used in the server details to update the order.

Listing 4.5 Representation of an order payload

```
{
    "status": "active",
    "created": "1588720456",
    "order": [
        {
            "product": "cappuccino",
            "size": "small",
            "quantity": 1
        },
        {
            "product": "croissant",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

For example, imagine that an order has the representation shown in listing 4.5. Suppose that the user wants to make a small amendment in this order, and change the size of the croissants from "medium" to "small". Although the user only wants to change one specific field of the whole payload, in a PUT request, you're required to send the whole payload back to the server, including fields that don't change. Because it's a PUT request, the server doesn't look for changes in the payload, and just uses the whole payload to update the details of the resource.

In a PATCH request, on the contrary, you're required to send only the fields that must be updated in the server. PATCH requests offer advantages in terms of optimization, since the payload sent to the server is smaller. It also allows for multiple concurrent updates to take place when each update targets a different property or field of the resource. However, in practice, most APIs expose the ability to perform updates on a resource with the PUT method, and in our specification for the orders API we will do the same.

To summarize, the specification for the orders API should include the following endpoints:

- **`POST /orders`**: endpoint to place orders.
- **`GET /orders`**: endpoint to retrieve a list of orders.
- **`GET /orders/{order-id}`**: endpoint to retrieve the details of a particular order.
- **`PUT /orders/{order-id}`**: endpoint to update the details of a particular order.
- **`DELETE /orders/{order-id}`**: endpoint to delete a particular order.

4.1.6 Using HTTP Status Codes to create meaningful HTTP responses

This section discusses best practices in the use of HTTP status codes in the responses from a RESTful API to a client. Status codes are part of the HTTP standard and they are used in HTTP responses in order to signal to the user what happened when processing a request. When properly used in the context of APIs, HTTP status codes can help us deliver meaningful responses to the users of our APIs. Status codes fall into the following five categories:

- **100 group**: used to signal that an operation is in progress.
- **200 group**: used to signal that a response is successful.
- **300 group**: used to signal that a resource has been moved to a new location.
- **400 group**: used to signal that something was wrong with the request.
- **500 group**: used to signal that there was an internal server error while processing the request.

The full list of HTTP status codes is long[5], and enumerating them one by one wouldn't do much to help us understand how we can best use them in the context of our responses. Instead, let's pick up our API specification where we left it in the previous section, and consider some of the most meaningful status codes that we could send back to our API consumers. We'll walk through each of the endpoints that we defined in the previous section, and consider different scenarios which require different HTTP status codes.

When thinking about how we can use HTTP status codes to deliver better responses to our API clients, we first have to distinguish between successful responses and non-successful responses. For each of the endpoints that we defined in the previous section, good HTTP status codes are the following:

- `POST /orders`: HTTP status code **201 (Created)**, which signals that a resource has been created.
- `GET /orders`: HTTP status code **200 (OK)**, which signals that the request was successfully processed.
- `GET /orders/{order_id}`: HTTP status code **200 (OK)**, which signals that the request was successfully processed.
- `PUT /orders/{order_id}`: HTTP status code **200 (OK)**, which signals that the request was successfully processed.
- `DELETE /orders/{order_id}`: HTTP status code **204 (No Content)**, which signals that the request was successfully processed but no content is delivered in the response. Contrary to all other methods, a DELETE request doesn't require a response with payload, since, after all, we are instructing the server to delete the resource. Therefore a 204 (No Content) is a good choice for this type of HTTP request. Alternatively, a 200 (OK) status code is also acceptable.

That's all good for successful responses, but how about error responses? What kind of errors can we encounter in our server while processing requests, and what kind of HTTP status codes might be most appropriate for them? We should distinguish two groups of errors:

[5] There are numerous resources in the Internet that you can use to check the list of HTTP status codes. Once website that I find particularly useful is https://httpstatuses.com/.

- Errors made by the user when sending the request, for example, due to a malformed payload, or due to the request being sent to an inexistent endpoint. This type of error is usually addressed with an HTTP status code in the 400 group.
- Errors unexpectedly raised in the server while processing the request, typically due to a bug in our code. This type of error is usually addressed with an HTTP status code in the 500 group.

Let's talk about each of these error types in more detail.

USING HTTP STATUS CODES TO REPORT ERRORS IN THE REQUEST

An API client can make different types of errors when sending a request to our server. Perhaps the most common type of error within this category has to do with malformed payloads or invalid URL query parameters (for the difference between these two concepts, see section 4.4 of this chapter). For example, let's say that, in order to place an order, our API expects a POST request on the /orders URL path with a payload like this:

```
{
    "order": [
        {
            "product": "mocha",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

That is, we expect the user to send us a list of elements, where each element represents an item of the order. Each item is described by the following keywords:

- product: it identifies the product that the user is ordering.
- size: it identifies the size that applies to the ordered product.
- quantity: it tells us how many items of the same product and size the user wishes to order.

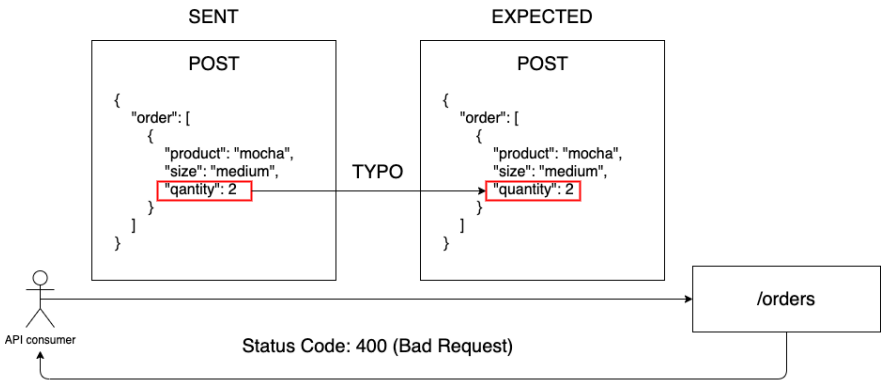


Figure 4.11 When an API client sends a malformed payload, for example due to a typo, the server responds back with a 400 (Bad Request) status code

When sending this payload, an API client can make a typo in one of the keywords, for example it could say "qantity" for "quantity", or it could send the wrong keywords, for example "amount" instead of "quantity". This kind of error happens very frequently in the world of API integrations, and we need to be prepared to respond adequately to our API clients, with meaningful responses and HTTP status codes. A very suitable HTTP status code to describe this error is **400 (Bad Request)**, which signals to the user that something was wrong with the request and therefore it couldn't be processed. Figure 4.11 illustrates this scenario.

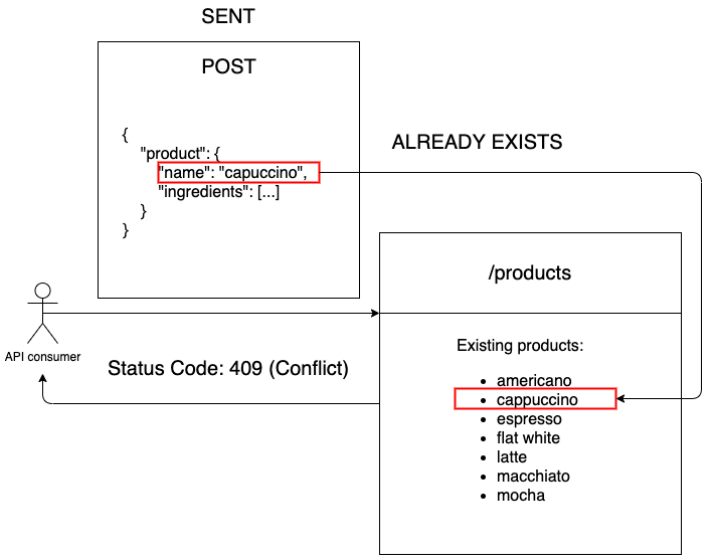


Figure 4.12 When an API client sends a request to create a resource which already exists, the server responds with a 409 (Conflict) status code

Sometimes the payload is correct, but it represents a resource which already exists and therefore would cause duplication in our records. This error could happen on the products API exposed by the products microservice, which we defined in chapter 3. The products microservice manages the catalogue of products of the CoffeeMesh. Products are identified by name, so we cannot have any two products with the same name. Any request to the API in order to create a product with an existing name should receive an error response indicating that the resource already exists. The HTTP status code **409 (Conflict)** is very suitable for these situations, as it's used to signal to the client that a request couldn't be processed because it creates a conflict with an existing resource. Figure 4.12 illustrates this scenario.

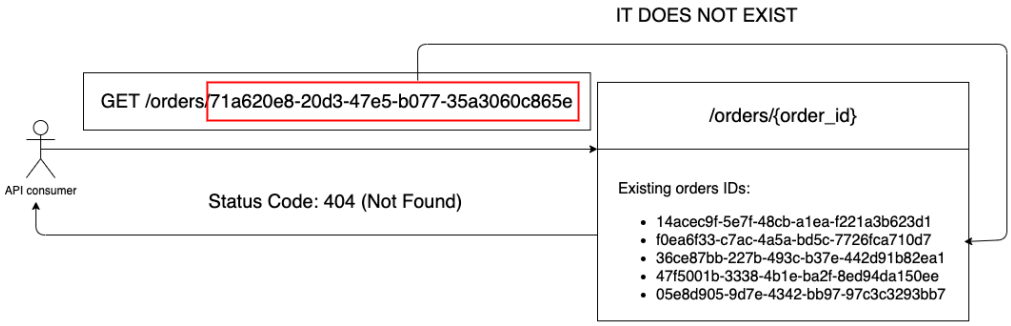


Figure 4.13 When an API makes a request on the URI for a resource that doesn't exist, the server responds with status code 404 (Not Found)

Another type of error that happens frequently when an API client sends a request to the server is when they call an inexistent endpoint. We know that a GET request on the `/orders/{order_id}` URL path serves to request the details of a specific order. If a user makes a typo when building the URI for the resource, for example by writing /orders/8e instead or /orders/8 in order to represent the order with ID 8, our server won't be able to find the resource requested by the user, and we should send back a meaningful HTTP status code signaling what happened. A very suitable HTTP status code for this scenario is **404 (Not Found)**, which is used to signal to the user that the requested resource is not available or couldn't be found. Figure 4.13 illustrates this scenario.

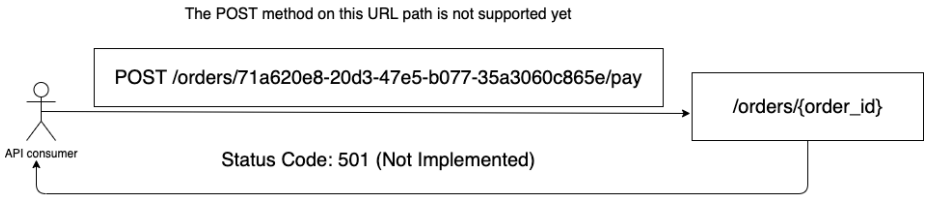


Figure 4.14 When an API client makes a request on a URL path with an HTTP method which will be exposed in the future but hasn't been implemented yet, the server responds with a 501 status code (Not Implemented)

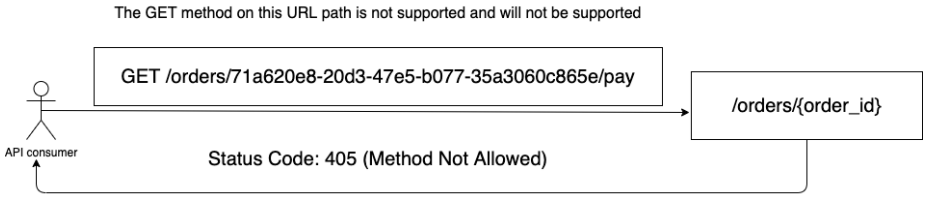


Figure 4.15 When an API client makes a request on a URL path with an HTTP method which is not supported and will not be supported, the server responds with a 405 status code (Method Not Allowed)

Another common error that API clients can make is sending a request on a specific URL path with an HTTP method which is not supported. For example, if a user sent a PUT request on the /orders endpoint, we should signal to them that the PUT method is not supported on that URL path. There are two HTTP status codes that we can use to address this situation. We can return a **501 (Not Implemented)** if the method hasn't been implemented but will be available in the future (i.e. we have a plan to implement it), or we can use **405 (Method Not Allowed)** if the method is not available and we don't have a plan to implement it. Figures 4.14 and 4.15 illustrate these scenarios.

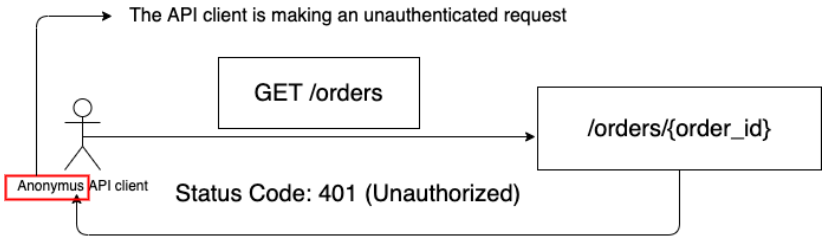


Figure 4.16 When an API client makes an unauthenticated request on an endpoint which requires authentication, the server responds with a 401 (Unauthorized) status code

Two additional errors that an API client can frequently make in their requests have to do with authentication and authorization. When a client makes a request to an authenticated endpoint without having authentication credentials, we should signal to them that they should first authenticate. A suitable HTTP status code to signal this error is **401 (Unauthorized)**, which is used to signal that the user hasn't been authenticated to access the endpoint. This status code is typically used when the client is making a request without required authentication credentials, or with expired credentials. This scenario is illustrated in figure 4.16.

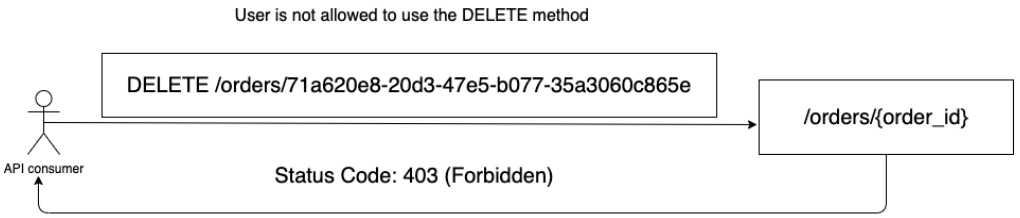


Figure 4.17 When an authenticated user makes a request using an HTTP method they're not allowed to use, the server responds with a 403 (Forbidden) status code

A related error is when a user is correctly authenticated, but they try to access an endpoint or a resource which they are not authorized to access. For example, for the orders API we have established that the `DELETE /orders/{order_id}` endpoint is not open to

normal users of the application, as it's used to delete records from our system. Therefore, any user sending a request to that endpoint should get a response informing them that they're not authorized to perform such action. A suitable HTTP status code for this scenario is **403 (Forbidden)**, which signals that the user doesn't have permissions to access the requested resource or to perform the requested operation. Figure 4.17 illustrates this scenario.

USING HTTP STATUS CODES TO REPORT ERRORS IN THE SERVER

The second group of errors we need to consider are those which are generated in the server due to a bug in our code or to a limitation in our infrastructure. The most common type of error within this category is when our application crashes unexpectedly due to a bug. In those situations, normally our web framework, such as Django or Flask, takes care of catching the application error and returning a payload with a **500 (Internal Server Error)** status code to the client.

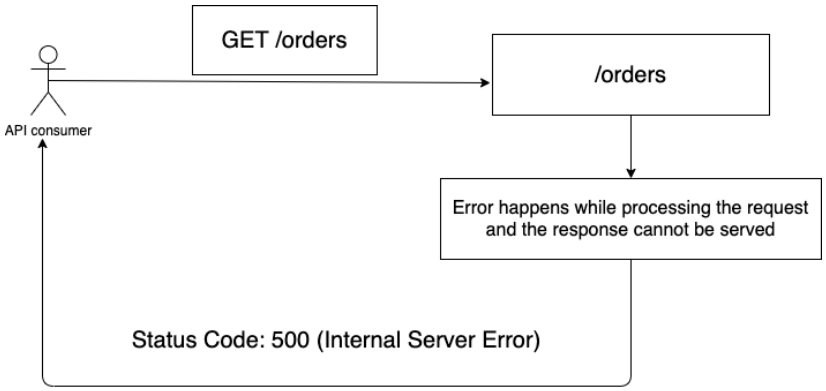


Figure 4.18 When the code processing a request raises an exception and therefore cannot serve the request, the server responds to the user with a 500 (Internal Server Error) status code

A different type of exception within this group happens when our application becomes unavailable or unable to service requests. This scenario can happen when the server is overloaded or down for maintenance, and we need to be able to let the user know about this situation by sending an informative payload together with a meaningful HTTP status code. We distinguish two situations in this case:

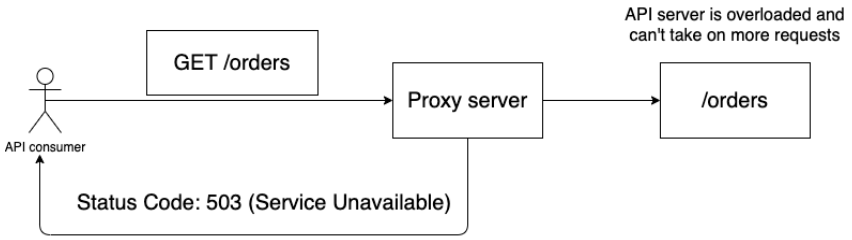


Figure 4.19 When the API server is overloaded and can't take on more requests, a proxy server responds to the client with a 503 (Service Unavailable) status code

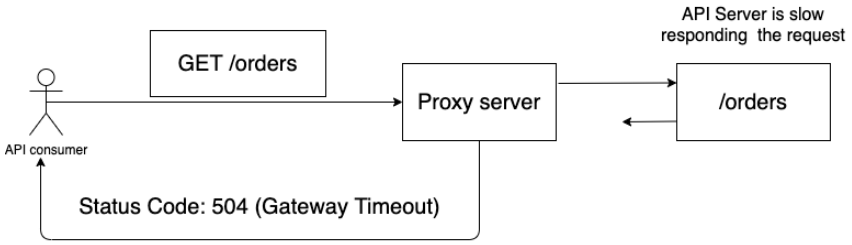


Figure 4.20 When the API server is very slow responding to the request, a proxy server responds to the client with a 504 (Gateway Timeout) status code

- When the server is unable to take on new connections. In this situation, we should have a proxy in front of the server which allows us to respond to the user with a **503 (Service Unavailable)** status code, which signals that the server is overloaded or down for maintenance and cannot service additional requests, as illustrated in figure 4.19. This state is temporary.
- When the server can take on new connections, but it takes too long to respond to the request. This situation can be handled by a proxy in front of our server, which can respond to the user with a **504 (Gateway Timeout)** status code, as shown in figure 4.20.

4.1.7 Returning meaningful responses to the API clients

In this section, we consider best practices for building meaningful responses for the clients of our APIs. We pick this up where we left it in the previous section, and we continue to extend the specification for our API by defining the payloads that will go into our responses. In this section, we focus mostly on the payloads that we should include in successful responses. Most error responses should include an error keyword pointing to a string containing a message that explains why the client is getting an error. For example, for a 404 response, which is generated when the resource requested by the user cannot be found in the server, we can return the following payload:

```
{
    "error": "Resource not found”
}
```

:For successful responses, we need to consider what the intent of an endpoint is in order to devise a meaningful response payload for it. In this section, we only look at representative examples of the payload for the response of each endpoint. In the following section of this chapter, we learn to express such payloads in a specification format. Let’s consider each endpoint one by one.

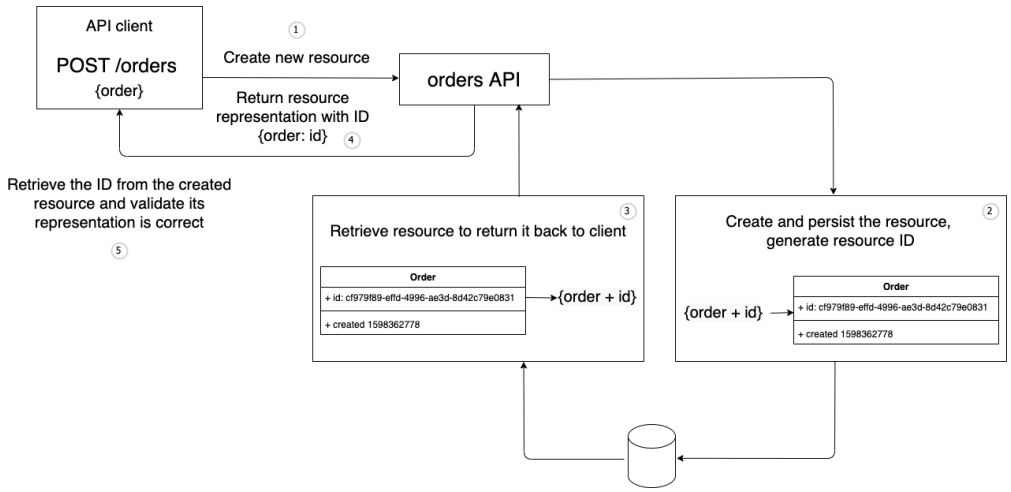


Figure 4.21 When an API client sends a POST request to create a new resource, the server responds back with a full representation of the resource just created together with the resource ID

`The POST /orders` endpoint creates a new order. Among other things, the process of creating a new order involves generating an ID for it, which we can later use to retrieve details about the order or check its status. Therefore, in the response for this endpoint, we should include a payload containing the ID of the recently created order. In fact, for POST endpoints it’s a good practice to return a full representation of the resource. This payload serves as a validation for the client that the resource was correctly created, as shown in figure 4.21. Therefore, the `POST /orders` endpoint should return a payload like the one shown in listing 4.6.

Listing 4.6 Example of response payload for the POST /orders endpoint

```
{
    "id": "31f134ef-aecd-4319-b693-9bf26f403a7a",
    "status": "active",
    "created": "1588720456",
    "order": [
        {
            "product": "cappuccino",
            "size": "small",
            "quantity": 1
        },
        {
            "product": "croissant",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

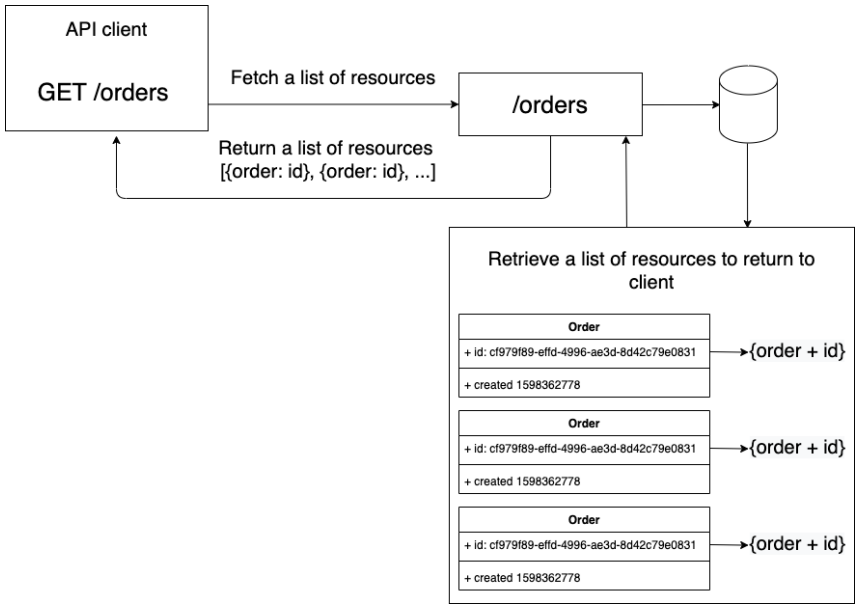


Figure 4.22 When an API client makes a GET request on the /orders URL path, the server responds with a list of orders, where each order object contains full details about the order

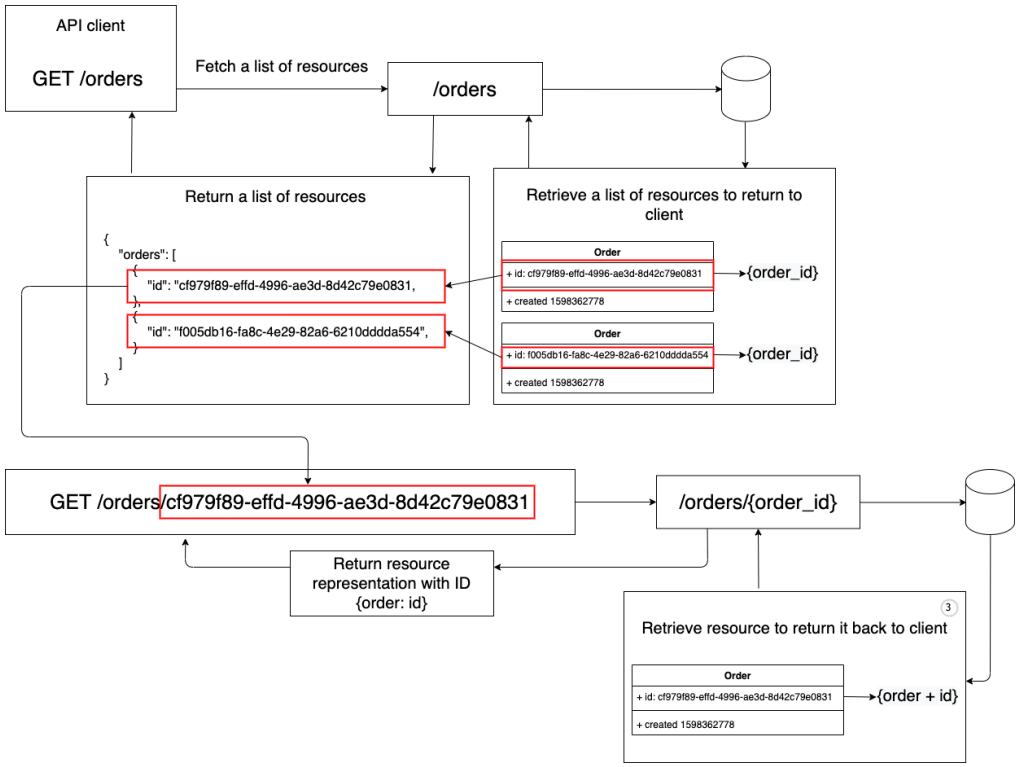


Figure 4.23 When the API client makes a GET request on the /orders URL path, the server responds with a list of order IDs, which then the client uses to retrieve the details of each order on the GET /orders/{order_id} endpoint

The response for the `GET /orders` endpoint is a list of orders. This list can contain full representations of each of the orders in the list, as shown in figure 4.22. However, in order to obtain a full representation of an order, we already have the `GET /orders/{order_id}` endpoint. It might seem like we are duplicating functionality if both endpoints return full representations of the orders. Alternatively, the `GET /orders` endpoint could return only a list of IDs, which we then can use to fetch full details from the `GET /orders/{order_id}` endpoint, one by one, as illustrated in figure 4.23. From an optimization point of view, however, it seems like waste to force the client to engage in multiple cycles of requests in order to fetch the full details of a list of orders. Which approach makes more sense then? The answer is: it depends on the use case. If the `GET /orders` endpoint is going to be used by a UI client in order to display details about recent orders made by a user, then it makes sense to serve all the details needed by the client at once. In such scenario, the response payload for the `GET /orders` endpoint would look like listing 4.7. If, instead, we wanted to return only a list of IDs through the `GET /orders` endpoint, the response payload would look like listing 4.8.

Listing 4.7 Example of response payload for the GET /orders endpoint with full details of each order

```
{
    "orders": [
        {
            "id": "31f134ef-aecd-4319-b693-9bf26f403a7a",
            "status": "active",
            "created": "1588720456",
            "order": [
                {
                    "product": "cappuccino",
                    "size": "small",
                    "quantity": 1
                },
                {
                    "product": "croissant",
                    "size": "medium",
                    "quantity": 2
                }
            ]
        },
        {
            "id": "f005db16-fa8c-4e29-82a6-6210dddda554",
            "status": "complete",
            "created": "1581978125",
            "order": [
                {
                    "product": "espresso",
                    "size": "big",
                    "quantity": 1
                },
                {
                    "product": "mocha",
                    "size": "medium",
                    "quantity": 2
                },
                {
                    "product": "almond cake",
                    "size": "small",
                    "quantity": 3
                }
            ]
        },
        {
            "id": "e3448371-41f2-42fe-b637-d35823eea8bd",
            "status": "active",
            "created": "1579299725",
            "order": [
                {
                    "product": "latte",
                    "size": "medium",
                    "quantity": 1
                }
            ]
        }
    ]
}
```

Listing 4.8 Example of response payload for the GET /orders endpoint showing only the IDs of each order

```
{
    "orders": [
        {
            "id": "31f134ef-aecd-4319-b693-9bf26f403a7a",
        },
        {
            "id": "f005db16-fa8c-4e29-82a6-6210dddda554",
        },
        {
            "id": "e3448371-41f2-42fe-b637-d35823eea8bd",
        }
    ]
}
```

As we have just stated, the primary purpose of the `GET /orders/{order_id}` endpoint is to return the full representation of an order. Picking the ID of one of the orders in the previous listing, for example `31f134ef-aecd-4319-b693-9bf26f403a7a`, a GET request on `the /orders/31f134ef-aecd-4319-b693-9bf26f403a7a` endpoint will return the following payload:

Listing 4.9 Example of response payload for the GET /orders/{order_id} endpoint

```
{
    "orders": [
        {
            "id": "31f134ef-aecd-4319-b693-9bf26f403a7a",
            "status": "active",
            "created": "1588720456",
            "order": [
                {
                    "product": "cappuccino",
                    "size": "small",
                    "quantity": 1
                },
                {
                    "product": "croissant",
                    "size": "medium",
                    "quantity": 2
                }
            ]
        }
    ]
}
```

A request on the `PUT /orders/{order_id}` endpoint should also return a full representation of the order, exactly as we did in the example for the `POST /orders` endpoint. The full representation of the order serves as validation for the API client that the resource as updated as expected. If we had a PATCH endpoint in the API, the response for such endpoint should follow the same guideline: return a full representation of the resource for validation in the client.

Finally, the response for the `DELETE /orders/{order_id}` endpoint shouldn't contain any payload. As we stated in the previous section, a typical HTTP status code for this endpoint is 204 (No Content), which means the response shouldn't contain any payload. Returning a response without payload makes sense, since the intention of a DELETE endpoint is to remove the resource from our records, and therefore there are no further representations of said resource that the server can return to the client.

4.4 Request payloads and URL query parameters

This section discusses the difference between request payloads, also called data, and URL query parameters. We explain why this difference is important, and when we should use payloads versus when we should use URL query parameters. Payloads and URL query parameters represent different elements of an HTTP request and they're used for different purposes.

4.1.8 Using payloads to send data to the server

In a RESTful API, some endpoints allow us to send data to the server in the form of a payload in order to create a new resource, or in order to update an existing resource. The `POST /orders` and the `PUT /orders/{order_id}` endpoints from the orders API are examples of endpoints which require us to send such payloads. The details about the resource we want to create or update are the data that goes into the payload (see Figure 4.9). A payload represents the body of an HTTP request, and importantly, we should only include payloads in POST, PUT, and PATCH endpoints. In a RESTful API, data is typically represented by a JSON document. For example, if we want to create an order for a cappuccino coffee in big cup, we may send the following payload to the `POST /orders` endpoint:

```
{
    "order": [
        {
            "product": "cappuccino",
            "size": "big",
            "quantity": 1
        }
    ]
}
```

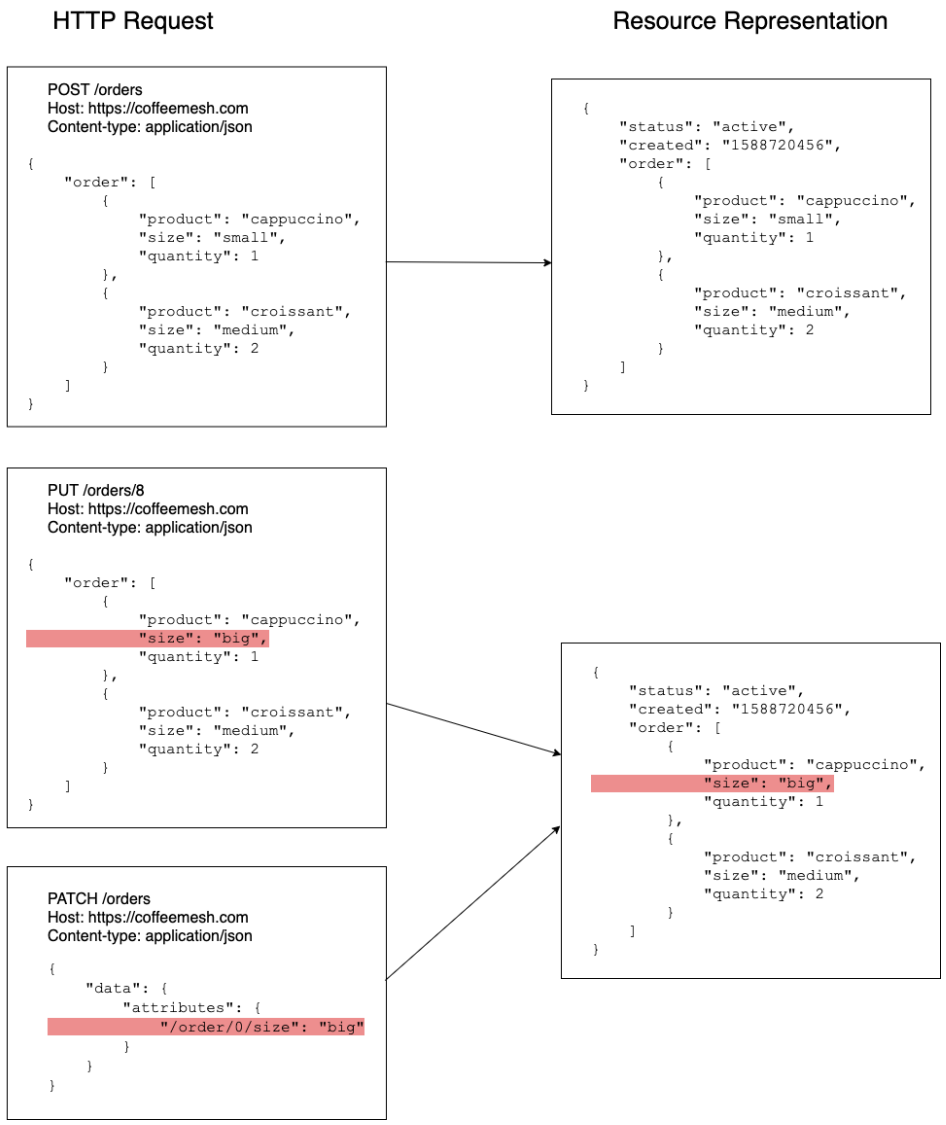


Figure 4.24. In a REST API, every endpoint which alters the representation of a resource must require clients to send the new representation in the body payload of an HTTP request. In this illustration, the `POST /orders` endpoint creates a new resource, and therefore the client sends a representation of that resource in the payload of the HTTP request. The `PUT /orders/{order_id}` endpoint update the representation of the resource, and therefore the client sends a new representation of the resource in the payload of the HTTP request. The `PATCH /orders/{order_id}` endpoint updates specific attributes of the resource, and therefore the client sends a representation of the attributes that must change in the payload of the HTTP request.

4.1.9 Using URL query parameters to filter resources

Let's now talk about URL query parameters and how, why, and when you should use them to retrieve lists of resources. Some endpoints, such as the GET /orders endpoint of the orders API, allow us to retrieve a list of resources. When an endpoint returns a list of resources, it's often possible to include filters in the URL which allow us to constrain the items that we get in the list. For example, when retrieving a list of orders through the GET /orders endpoint, we may want to be able to limit the results to only the five most recent orders, or we may be able to list only those orders that were cancelled in the past. How can we accomplish this?

URL query parameters are key-value pairs which form part of a URL but are separated from the URL path by a question mark. We can chain multiple query parameters within the same URL by separating them with ampersands. Since query parameters are collections of key-value pairs, order shouldn't matter when we include them in the URL. For example, if we want to form a URL with query parameters for the GET /orders endpoint restricting the results to the last five cancelled orders, we may write something like this

```
GET /orders?cancelled=true&limit=5
```

It gets complicated pretty soon. For example, say we want to retrieve a list of products whose rating is greater than or equal to 4. How can build such a filter into the URL using query parameters? We could do the following:

```
/products?rating=4+
```

Or

```
/products?rating=>4
```

This solution is, however, not very robust, because it

- forces our URL parser to be very smart when retrieving the value of the rating parameter, and therefore it makes the validation of this parameter a lot more difficult.
- mixes types: a number representing the rating (4), and a string representing the comparison operator we want to apply on the number (>). Mixing types makes it more difficult to interpret the values.
- neither the `+` symbol nor `>` symbol can be directed represented in an URL. The `+` symbol gets converted to `%2B`, and the `>` symbol gets converted to `%3E` when you apply URL encoding on them. This makes the URL less readable and user-friendly.

A better approach is to break down the operator and the rating value into two query parameters: `rating` and `rating_operator`. The rating operator will admit any of the following values:

- `e` for equal to. This is the default value in case the parameter is missing.
- `gt` for greater than.
- `gte` for greater than or equal to.
- `lt` for less than.
- `lte` for less than or equal to.

The query would now look like this:

```
/products?rating=4&rating_operator=gte
```

When you design your API endpoints, it's very important that you maintain and apply consistently the distinction between query parameters and payload and use them appropriately, as described in this section. Unfortunately, it's not unusual to see APIs where query parameters are requested in a payload on a GET request, as shown in listing 4.10.

Listing 4.10 Example of GET request with filters included in a payload

```
GET /products
{
    "filters": [
        {
            "rating": 4
        },
        {
            "rating_operator": "gte"
        }
    ]
}
```

This approach is flawed, since the HTTP protocol specification discourages the use of payloads in GET requests. The HTTP Working Group's "HTTP Semantics" (https://tools.ietf.org/html/draft-ietf-httpbis-semantics-10) defines the semantics of the HTTP methods. This document is subject to changes and updates, but since its inception, the specification has discouraged the use of payloads in GET requests. As the specification states,

> ***A client SHOULD NOT generate a body in a GET request. A payload received in a GET request has no defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling attack (https://tools.ietf.org/html/draft-ietf-httpbis-semantics-07#section-7.3.1).***

This means that there is nothing preventing you from implementing a GET endpoint which accepts a payload, but such payload should be meaningless, and the server has no obligation to be able to interpret it. Furthermore, as the specification points out, some web framework implementations may completely prevent you from implementing GET endpoints that accept payloads. The Internet Draft "Building Protocols with HTTP," written by M. Nottingham from the HTTP Working Group, makes this observation more explicit:

> ***Finally, note that while HTTP allows GET requests to have a body syntactically, this is done only to allow parsers to be generic; as per [I-D.ietf-httpbis-semantics], Section 7.3.1, a body on a GET has no meaning, and will be either ignored or rejected by generic HTTP software (https://tools.ietf.org/html/draft-ietf-httpbis-bcp56bis-09#section-4.5.1).***

4.5 Summary

- Representational State Transfer (REST) is an architectural style for web APIs which provides a number of guidelines for building scalable web applications and meaningful communication processes between the API consumer and the server.
- RESTful APIs are characterized by six fundamental constraints:
    - Client-server architecture
    - Statelessness
    - Cacheability
    - Layered system
    - Code on demand (optional)
    - Uniform interface
- In addition, Hypermedia as the Engine of Application State (HATEOAS) suggests that RESTful APIs should allow a client to navigate the whole API by providing lists of related links in the payloads of the responses.
- CRUD defines the basic operations that any system that deals with data should be able to perform:
    - Create
    - Read
    - Update
    - Delete
- CRUD is a useful concept when designing RESTful APIs and the type of actions and capabilities that we want to expose to our users.
- RESTful web APIs are built on top of the HTTP protocol, and leverage all of the resources that the HTTP specification provide in order to create meaningful communication between a client and a server.
- The most relevant HTTP methods in the context of RESTful APIs are:
    - GET for fetching resources
    - POST for creating resources
    - PUT for replacing resources
    - PATCH for updating resources
    - DELETE for deleting resources
- HTTP responses from a RESTful API should include meaningful HTTP status codes, such as 200 for the successful response to a GET request, or 201 for the successful response to a POST request.
- Data for creating or updating resources in the server must be sent in the payload of an HTTP request.
- Parameters for filtering resources in a GET request must be included as URL query parameters.

5

Producing a REST API specification

This chapter covers:

- Why the JSON Schema is useful for defining API specifications
- Learning the Swagger/OpenAPI specification
- Defining the specification for the payloads in the requests and responses of the API
- Using generic components to refactor our schemas and write better and more readable and maintainable API specifications
- Generating API specifications with code

In this chapter, we put together all the concepts and ideas that we developed in chapter 4in order to write the specification for the orders API using the Swagger/OpenAPI standard. To do this, we first provide an overview of what JSON Schema is. JSON Schema is a specification format that can be used to define the structure of a JSON document, including the types and formats of the values within the document. In this chapter, we explain how we can leverage the concepts and resources that JSON Schema provides in order to define better and more readable schemas.

Swagger/OpenAPI is the most popular standard for describing RESTful APIs, and it provides a rich ecosystem of tools that we can use to validate and visualize our specifications. Additionally, as we will see in chapter 6 Python has numerous libraries built around the concepts of Swagger/OpenAPI, and those libraries will help us leverage the OpenAPI specification for our REST APIs to improve our development process.

In this chapter, we provide an overview of how a Swagger/OpenAPI specification is structured, explaining the minimum set of attributes that we should aim to provide in order to make our specification sufficiently informative for the consumers of our API. API endpoints constitute the core of the specification, so we pay particular attention to their definitions. We break down step by step the process of defining the schemas for the payloads of the API's

requests and responses, and we study different methods to refactor the schemas in order to produce more readable and maintainable specifications.

5.1 JSON Schema for API documentation

This section introduces the specification standard for JSON documents called JSON Schema, and explains how we can leverage its capabilities in order to produce our API specifications. OpenAPI uses an extended subset of the JSON Schema specification. In this section, we'll learn the fundamentals of JSON Schema as they relate to OpenAPI. JSON Schema is format specification that can be used to define the structure of a JSON document and the types of its attributes. JSON Schema is useful for documenting interfaces which use JSON to represent data, and to validate that the data being exchanged is correct and has the right format according to the specification. The JSON Schema's specification is under active development in the Internet Engineering Task Force (IETF) website, with the latest version being the Internet Draft 2019-09[1]. Data exchanged over HTTP following a JSON Schema is a specification uses the media type "application/schema+json".

JSON Schema allows us to be very explicit with respect to the data types and formats that both the server and the client should expect from a payload. This is fundamental for the integration between the API provider and the API consumer, since it lets us know how to parse the payloads and how to cast them into the right data types in our runtime when consuming the data.

JSON Schema supports the following basic data types:

- `string` for character values.
- `number` for integer and decimal values.
- `object` for associative arrays (aka dictionaries in Python).
- `array` for collections of other data types (aka lists in Python).
- `boolean` for `true` or `false` values.
- `null` for uninitiated data.

A JSON Schema specification usually defines an object with certain attributes or properties. A JSON Schema object is represented by an associative array of key-value pairs, with each key representing a named property or attribute of the object, and the value representing the specification about the attribute. The specification for each attribute is itself another object with one required attribute: the type of the attribute. What this apparently complex explanation means is that a JSON Schema specification usually looks like this:

```
{
    "status": {    #A
        "type": "string"    #B
    }
}
```

#A Each property in a JSON Schema specification comes as a key whose values are the descriptors of the property

[1] A. Wright, H. Andrews, B. Hutton, "JSON Schema: A Media Type for Describing JSON Documents", September 16, 2019 (https://tools.ietf.org/html/draft-handrews-json-schema-02). You can follow the development of JSON Schema and contribute to its improvement by participating in its repository in GitHub: https://github.com/json-schema-org/json-schema-spec. Checkout also the website for the project: https://json-schema.org/.

#B The minimum descriptor necessary for a property is the type. In this case, we specify that the status property is a string

In this specification, we are defining the schema of an object with one attribute named `status`, which has type string. Attributes can also be objects, as in the following example:

```
{
    "status": {
        "type": "string"
    },
    "order": {
        "type": "object",    #A
        "properties": {    #B
            "product": {    #C
                "type": "string"
            },
            "size": {
                "type": "string"
            },
            "quantity": {
                "type": "integer"
            }
        }
    }
}
```

#A We can define the type of a property as an object. We normally do this when the property we are describing has nested properties
#B The properties of an object are described under the properties keyword
#C Each property of a JSON Schema object is described just in the same way as a top-level property. At a minimum, you'll want to specify the type of the property. The property of a JSON Schema object can be of any type, including an object

In this case, we are defining an object with two properties: `status` and `order`. The `order` attribute is an object that represents an order, and is declared as such by the type. You can think of objects in JSON Schema as similar to classes in Python. Classes in Python have attributes that help you represent the properties of an object, and the same happens with JSON Schema objects. Since it's an object, the `order` attribute also has properties, which in this case are defined under the `properties` attribute. Each property also requires its own type specification. A JSON document that complies with the specification is the following:

```
{
    "status": "active",
    "order": {
        "product": "coffee",
        "size": "big",
        "quantity": 1
    }
}
```

As you can see, each of the properties described in the specification is used in this document, and each of them has the expected type.

A property can also represent an array of objects. In the following example, the order object represents an array of orders:

```
{
    "status": {
        "type": "string"
    },
    "order": {
        "type": "array",
        "items": {    #A
            "type": "object",
            "properties": {
                "product": {
                    "type": "string"
                },
                "size": {
                    "type": "string"
                },
                "quantity": {
                    "type": "integer"
                }
            }
        }
    }
}
```

#A When the type of a property is an array, we need to provide the specification for the properties that go inside the array. This is done by using the items keyword, under which we can define the type and any other necessary attributes of the properties in the array. In this case, we specify that each property in the array is an object, and we specify the properties of such object

In this case, the `order` property points to a property which is defined as an array. Array types require an additional property in their schema, which is the `items` property. The items property defines the type of each of the elements contained in the array. In this case, each of the elements in the array is an object that represents an order.

An object can have any number of nested objects. However, when too many objects are nested within the same definition, the levels of indentation can grow large and make the specification difficult to read. To avoid this situation, JSON Schema allows us to use JSON Pointers, which is a special syntax that allows us to point to another object definition within the same specification. We can extract the definition of each order object within the array of the `order` property in the previous example, and use a JSON pointer to specify the type of each element within the array:

```
{
    "OrderItem": {
        "type": "object",
        "properties": {
            "product": {
                "type": "string"
            },
            "size": {
                "type": "string"
            },
            "quantity": {
                "type": "integer"
            }
        }
    },
```

```
    "Order": {
        "status": {
            "type": "string"
        },
        "order": {
            "type": "array",
            "items": {
                "$ref": '#/OrderItem'    #A
            }
        }
    }
}
```

#A When the specification of a property is an object that we have already defined somewhere else in the document, we can provide a pointer to that definition using a JSON pointer. A JSON pointer is a specific syntax in JSON Schema that allows us to represent the path to a specific element within the schema. JSON pointers are used in combination with the $ref keyword.

JSON pointers use the special keyword $ref, and the special JSON Path syntax to point to an attribute of the schema. In JSON Path syntax, the root of the document is represented by the hashtag symbol (#), and the relationship of nested properties is represented by forward slashes (/). For example, if we wanted to create a pointer to the size property of the `OrderItem` model, we would use the following syntax: `'#/OrderItem/size'`.

Using JSON Pointers, we can refactor our specification by extracting common schema definitions into their own models, and have other schemas point to them. This helps us to avoid duplication and to keep the specification clean and succinct.

In addition to being able to specify the type of an attribute, JSON schema also allows us to specify the format of the attribute. We can develop our own custom formats, or use JSON Schema's built-in formats. Format specifications are particularly useful with string types that we expect in a specific format. For example, for an attribute representing a date, we can specify that the date should have a `date` format. date is a built-in format supported by JSON Schema which represents an ISO date with reference only to the year, month, and day, e.g. 2025-05-21. Here's an example of such a specification:

```
{
    "created": {
        "type": "string",
        "format": "date"
    }
}
```

For a full list of the built-in data formats supported by JSON Schema, please refer to appendix B.

The examples in this section have been provided in JSON format. However, JSON Schema specification don't need to be written in JSON necessarily. In fact, it's more common to write them in YAML format, as it's more readable and easier to understand. The OpenAPI specification for the orders API that we build in this chapter is written in YAML format.

5.2 Documenting the API with Swagger/OpenAPI

In this section, we introduce the Swagger/OpenAPI standard, and we learn to produce an API specification. We put together all the concepts and ideas that we have developed so far around the orders API, and we formalize them into a standard specification that any OpenAPI parser can understand. We use the latest version of OpenAPI (3.0.3). Appendix A contains the full specification for the orders API, and you can refer to it at any time to follow along with the explanations in this chapter.

Swagger/OpenAPI is a standard specification format for documenting RESTful APIs. Swagger/OpenAPI allows us to describe in detail every element of an API which a client needs to know to be able to interact with it, including its endpoints, the format of its request and response payloads, its security schemes, and so on. It was created in 2010 under the name of Swagger as an open-source specification format for describing RESTful web APIs. Over time, this framework grew in popularity, and in 2015, the Linux Foundation and a consortium of major companies sponsored the creation of the OpenAPI initiative, a project aimed at improving the protocols and standards for building RESTful APIs. Today, Swagger/OpenAPI is by far the most popular specification format used to document RESTful APIs, and a vast array of tools that we can use to visualize and validate our specifications have been developed, as well as libraries that allow us to leverage the specification during our software development process.

5.2.1 Anatomy of a Swagger/OpenAPI file

This section explains what the general structure of a Swagger/OpenAPI specification looks like. We provide a high-level overview of each section before delving into more detail in the following sections. A Swagger/OpenAPI specification contains everything that the consumer of an API needs to know in order to be able to interact with it. A typical Swagger/OpenAPI is structured around the following sections:

1. **Metadata**: this section contains general information about the specification that we are providing and normally goes at the top of the file. The most important piece of information in this section is the version of OpenAPI that we are using for the spec. The OpenAPI version is crucial because validation tools will validate our specification against that version, and API parsers will use it in order to parse its structure. Other pieces of information that go into this section are the title of the API, a short description, and the version of our API.
2. **Servers**: The servers section of an OpenAPI specification contain a list of URLs where the API is available. Understand "server" here in an abstract way, i.e. not as the address of a physical server, but as the DNS record which points to this particular API. In a modern architecture, behind the DNS record you will have load balancers which point to a varying number of servers. However, this complexity is not the concern of your API consumers, and should be encapsulated behind a single endpoint. If you list more than one URL in the servers section of your API specification, each entry will point to a different environment, such as the production and the staging environments.

3. **Paths**: this section describes the endpoints exposed by the API. Each path description includes the HTTP methods it implements, the expected payloads and allowed parameters, if any, and the format of the response, including the expected status codes. This is the most important part of the specification file, as it defines the interface for the API, and it's the section that consumers will be looking for in order to know what to expect from the API.
4. **Components**: the components section of a Swagger/OpenAPI specification allows you to define reusable schemas that you can reference across the file. A schema is a definition of the expected attributes and types in your request and response objects. OpenAPI schemas are defined using the JSON Schema specification.
5. **Authentication**: this section allows you to specify how your API endpoints are protected and how the consumer of the API has to form their requests so that they can authenticated and authorized by the server.

5.2.2 Documenting the API endpoints

In this section, we develop the specification for the endpoints of the orders API. The paths section of an OpenAPI specification describes the interface of your API. It lists the endpoints which are supported by the API, together with the HTTP methods they implement, the type of requests they expect, and the responses that they will return, including the status codes. Each path is an object whose attributes are the HTTP methods it supports. From the analysis that we have realized in previous sections of this chapter, we have established that the orders API contains the following endpoints:

- `POST /orders`: creates an order. It requires a payload with the details of the order to be created.
- `GET /orders`: returns a list of orders. It accepts URL query parameters which allow us to filter the orders to be returned.
- `GET /orders/{order_id}`: returns the details of a specific order. It doesn't accept request payloads or URL query parameters.
- `PUT /orders/{order_id}`: updates the details of a specific order. It requires a payload with the new details of the order. Since this is a PUT endpoint, the payload must contain all the details of the order.
- `DELETE /orders/{order_id}`: deletes a specific order. It doesn't accept request payloads or URL query parameters.

To this list of endpoints, we are going to add two additional endpoints which will allow the user to pay for the order and to track its status:

- `POST /orders/{order_id}/pay`: allows the user to pay for the order. It requires a payload with the payment details of the user. We use a the HTTP method POST to define this endpoint since it requires a payload, and because it triggers an action on the resource, but does not directly update the resource.
- `GET /orders/{order_id}/status`: allows the user to retrieve the status of the order. We use the HTTP method GET to define this endpoint since we are only retrieve details about the resource.

The high-level structure of these endpoints in the `paths` section of the OpenAPI specification for the orders API is the following:

```
paths:
  /orders:
    get:  # retrieves a list of orders made by the user
    post:  # creates a new order

  /orders/{order_id}:
    get:  # retrieves the details of an order
    put:  # updates the details of an order
    patch:  # updates the details of an order
    delete:  # deletes an order made by the user

/orders/{order_id}/pay:
    post:  # processes payment for the order

/orders/{order_id}/status:
    get:  # it retrieves the status of the order
```

Each of the HTTP methods in the above listing is in an object which contains all the information that the consumer of the endpoint needs in order to call it correctly. The information contained in this section is crucial for the correct integration between your server and your clients, so it pays off to make it as detailed as you can. For the GET /orders endpoint, we need to describe the parameters that the endpoint accepts, and for the POST and PUT endpoints we need to describe the payloads. We also need to describe the responses for each endpoint. Let's get on to it! In the following sections, we'll learn to build specifications for different elements of the API, starting with the URL query parameters.

DOCUMENTING URL QUERY PARAMETERS

In this section, we'll see how we can document the URL query parameters accepted by the GET /orders endpoint. We will allow users to filter orders by the following factors:

- `cancelled`: whether the order was cancelled or not. This value will be a Boolean.
- `limit`: it specifies the maximum number of orders that should be returned to the user. The value for this parameter will be a number.

Both `cancelled` and `limit` can be combined within the same request to filter the results, for example: `GET /orders?cancelled=true&limit=5`. This request is asking the server to return a list of 5 orders, all of which have been cancelled. The specification for the GET /orders endpoint, including the parameters, is as follows:

```
paths:    #A
  /orders:    #B
    get:    #C
      parameters:    #D
        - name: cancelled    #E
          in: query    #F
          required: false    #G
          schema:    #H
            type: boolean
        - name: limit
          in: query
          required: false
```

```
          schema:
            type: integer
```

#A We describe the paths of an API under the paths section of the specification document
#B Every path is an object where the key is the name of the path (in this case /orders) and the value are the properties that describe the path. The most important properties are the HTTP methods supported by the path. In this case we describe the GET method
#C The HTTP method supported by the path
#D If an HTTP method accepts parameters, either in the URL or in the path, we can describe them under the parameters property
#E The name of the property
#F The in descriptor allows us to specify where the parameter goes, either in the URL path or in the URL query parameters
#G We can specify whether the parameter is requried or not
#H The schema descriptor allows us to specify the type of the parameter

The definition of a parameter requires a name, which is the value we use to refer to it in the actual URL. We also need to specify what type of parameter it is. OpenaPI 3.0 distinguishes four types of parameters: path parameters, query parameters, header parameters, and cookie parameters. Header parameters are parameters that go in an HTTP Header field, while cookie parameters are parameters that go into a Cookie payload. Path parameters are part of the URL definition and are typically used to identify a resource. For example, in `/orders/{order_id}`, `order_id` is a path parameter which identifies a specific order. Query parameters are optional parameters that allow us to filter the results of an endpoint, such as `cancelled` and `limit` in the example that we are discussing now. The schema attribute of a parameter definition allows us to define the type of the parameter (Boolean in the case of `cancelled`, and number in the case of `limit`), and when relevant, the format of the parameter as well[2]. Now that we know how to describe URL query parameters, let's move on to something a bit more complex: request payloads.

DOCUMENTING REQUEST PAYLOADS

In this section, we see how we can document the request payload specifications of our API endpoints. Specifically, we'll work out the definitions for the payload definitions for the POST and PUT methods of our API. Let's start with the `POST /`orders method. In section 5.4, we established that the payload for the `POST /orders` endpoint looks like this:

```
{
    "order": [
        {
            "product": "cappuccino",
            "size": "big",
            "quantity": 1
        }
    ]
}
```

[2] To learn more about the date types and formats available in OpenAPI 3.0, head over to the following link: http://spec.openapis.org/oas/v3.0.3#data-types.

This payload contains an attribute `order` which points to an array of objects, each of each represents an item in the order. Each item is defined by the following three attributes and constraints:

- `product`: represents the type of product that the user is ordering.
- `size`: represents the size of the product. It can be one of the three choices: small, medium, and big.
- `quantity`: represents the amount of the product. It can be any integer number equal to or bigger than one.

Listing 5.1 shows how we represent the schema for this object in an OpenAPI specification.

Listing 5.1 Specification for the POST /orders endpoint

```
paths:
  /orders:
    post:
      requestBody:    #A
        required: true    #B
        content:    #C
          application/json:
            schema:    #D
              type: object
              properties:
                order:
                  type: array
                  items:
                    type: object
                    properties:
                      product:
                        type: string
                      size:
                        type: string
                        enum:    #E
                          - small
                          - medium
                          - big
                      quantity:
                        type: integer
                        required: false
                        default: 1    #F
```

#A We describe request payloads under the requestBody keyword
#B We can specify whether the payload is required
#C APIs can accept payloads in different formats or content types, such as JON and XML, and OpenAPI allows us to describe each of them. We describe each content type under the content keyword. Our API only accepts JSON payloads, so we include only one entry for application/json
#D The schema for the payload document itself goes under the schema keyword
#E We can constrain the values that a property can take by providing an enum
#F We can specify a default value for a property by using the default keyword

The request payload of a method is defined in the `content` attribute of the `requestBody` attribute of the POST method. With OpenAPI, we can specify payloads in different formats or

serialization methods. In this case, we are only allowing data to be sent in JSON format, which has a media type definition of `application/json`. The schema for our payload represents is an object with one property: `order`, which has type `array`. The items in the array are objects with three properties: the `product` property, with string type; the `size` property, with string type; and the `quantity` property, with integer type. In addition, we have defined an enumeration for the `size` property which specifies which values are accepted in this property: small, medium, and big. Finally, we also provide a default value of 1 for the `quantity` property. We give a default value to this property in order to be able to make it optional. That way, whenever a user sends a request containing an item without the `quantity` property, we can safely assume that they intend to order only one unit of that item.

REFACTORING SCHEMA DEFINITIONS TO AVOID REPETITION

In this section, we'll learn some strategies to refactor the schemas used in the specifications of the API endpoints to make sure we keep the API specification clean and readable. The current definition of the schema of the `Order` object does the job, but it's a bit long and it contains several layers of indentation. As a result, it's difficult to read, and that means that in the future it'll become difficult to extend and to maintain. We can do better than this by moving the schema definition of our payload into a different section of the API specification: the components section. As we explained in section 5.2.1, the components section of an API specification can be used to declare schemas that can be reused in different parts of the document, and we are going to do just that. Listing 5.2 shows our refactored specification.

Listing 5.2 Specification for the POST /orders endpoint using a JSON pointer to point to the payload schema

```
paths:
  /orders:
    post:
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Order'    #A

components:    #B
  schemas:
    Order:    #C
      type: object
      properties:
        order:
          type: array
          items:
            type: object
            properties:
              product:
                type: string
              size:
                type: string
                enum:
```

```
                - small
                - medium
                - big
            quantity:
              type: integer
              required: false
              default: 1
```

#A We use a JSON pointer to point to a schema that we define somewhere else in the document. JSON pointers are used in combination with the $ref keyword

#B Schema definitions go under the components section of the API specification, together with other common elements referenced through the API. In addition to schemas, components can be used to define other elements which are commonly used throughout the specification, such as parameters, security schemes, request bodies, and responses, among others[3]. Each of these elements goes under their own header. Schemas definitions go under the schemas keyword.

#C Each schema is an object where the key is the name of the schema and the values are the properties that describe it. We use the schema name to refer to it when we use a JSON pointer

This refactoring makes the API more readable. It allows us to keep the paths section of the document clean and focused on what it really has to do, which is to provide us with higher-level details about how to interact with the endpoint. The more specific details about the payload can be found in a section which is specially designed to accommodate this type of definitions: the components section of the API. All we have to do is refer to refer to the `Order` schema using a JSON pointer, which has the format `'#/components/schemas/Order'`.

The specification is looking good, but it can get still better. It's good to have payloads schemas defined within the components section of the document. Our definition of the `Order` schema is, however, a tad long, and it still contains several layers of nested definitions. If this specification grows in complexity, over time it'll become difficult to read and maintain. We can make it more readable by refactoring out the definition of the order property into its own independent schema. One added benefit of this strategy is that it allows us to reuse the schema for the item in other parts of the API, if we need to. Listing 5.3 shows the refactored schema definitions.

Listing 5.3 Schema definitions for OrderItem and Order

```
components:
  schemas:
      OrderItem:    #A
        type: object
        properties:
          product:
            type: string
          size:
            type: string
            enum:
              - small
              - medium
              - big
          quantity:
```

[3] See https://swagger.io/docs/specification/components/ for a full list of reusable elements that can be defined within the components section of the API specification.

```
            type: integer
            default: 1
      Order:
        type: object
        properties:
          order:
            type: array
            items:
              $ref: '#/OrderItem'    #
```

#A We introduce a new object to define the schema for each item within the order array
#B We use a JSON pointer to point to the schema of each object within the order array

Our schemas are looking good! The `Order` schema can be used to create an order or to update it, so we can reuse the same schema in the specification of the endpoint that we designed previously to update orders: the PUT `/orders/{order_id}` endpoint. Listing 5.4 shows the implementation of the PUT `/orders/order_id}` endpoint.

Listing 5.4 Specification for the PUT /orders/{order_id} endpoint

```
paths:    #A
  /orders:
    get:
      ...    #B

  /orders/{order_id}:    #C
    parameters:    #D
      - in: path    #E
        name: order_id    #F
        required: true    #G
        schema:
          type: string
          format: uuid    #H
    put:    #I
      requestBody:    #J
        required: true
        content:
          application/json:
            schema:
             $ref: '#/components/schemas/Order'
```

#A We include the new endpoint definition within the paths section of the API specification, together with all other endpoints we defined previously
#B We omit the specifications of the endpoints we defined previously to focus on the new endpoint
#C The PUT endpoint is defined under a new path: /orders/{order_id}. order_id is a parameter in the URL path, and therefore it's wrapped in curly braces
#D We define the URL path parameter directly under the URL path to make sure it applies to all HTTP methods defined within this path
#E The new parameter is part of the path
#F The name of the parameter. In the URL path, we use this name to indicate where exactly in the URL path the parameter goes
#G Since the parameter is part of the URL path, it's required
#H In addition to specifying the type of a property, we can also specify its format. The order_id parameter has the format of a universally unique identifier (UUID)

#I Definition of the HTTP method PUT for the current URL path
#J Specification for the request body of the PUT endpoint

The specification for the PUT `/orders/{order_id}` endpoint is similar to the POST /orders endpoint, but it also contains definitions for the URL parameter that we use to refer to the specific order that we are going to update. The definition of a path parameter is similar to the definition of a URL query string parameter. In this case, we are specifying that the `order_id` parameter is a string with a Universally Unique Identifier (UUID) format. A UUID is a long string with the form of a hash which is commonly used to define the IDs of records in distributed databases. In previous examples, we have used simple IDs, such as the integer 8, to refer to the ID of an order, but in real life you're more likely to use UUIDs. An example of UUID is `b4ce26c9-8c77-47a9-896a-8589a72431cb`. Now that we understand how to define the schemas for our request payloads, let's turn our attention to the schemas for our responses.

DOCUMENTING API RESPONSES

In this section, we'll see how we can document the responses of endpoints exposed by the orders API. We'll first define the payload for the `GET /orders/{order_id}` endpoint, and then we'll move on to produce a full specification of the API response as per the requirements of the OpenAPI standard.

The response of the `GET /orders/{order_id}` endpoint might look like this:

```
{
    "status": "active",
    "created": "1588720456",
    "order": [
        {
            "product": "cappuccino",
            "size": "small",
            "quantity": 1
        },
        {
            "product": "croissant",
            "size": "medium",
            "quantity": 2
        }
    ]
}
```

This payload contains details about the products ordered by the user, the time when the order was made in the form of a timestamp, and the status of the payload. We have decided to return the date of creation in the form of a timestamp because web APIs are programmable APIs (see section 1.2.1 in chapter 1 for more details about this), and every client which interacts with the API will be a programmable client with capabilities to translate the timestamp into the most suitable date format for the user.

The payload for the response in the `GET /orders/{order_id}` endpoint is very similar to the payload for the request in the POST and PUT endpoints that we defined previously. The response payload for the `GET /orders/{order_id}` endpoint only adds two new attributes: the "status" attribute and the "created" attribute. Therefore, in order to

define the schema for the response payload of the `GET /orders/{order_id}` endpoint, we can reuse the `OrderItem` schema previously defined. Listing 5.5 shows the definition of the `GetOrder` schema.

Listing 5.5 Definition of the GetOrder schema

```
components:    #A
  schemas:
    OrderItem:
      ...    #B

  GetOrder:    #C
    type: object
    properties:
      status:
        type: string
        enum:
          - active
          - completed
          - cancelled
      created:
        type: integer
        description: Date in the form of UNIX timestmap    #D
      order:
        type: array
        items:
          $ref: '#/components/schemas/OrderItem'    #E
```

#A The specification for GetOrder goes under the components section of the API specification, together with all other schemas we defined previously
#B We omit the schemas we previously defined to focus on the new changes
#C Definition of the GetOrder schema
#D We can include a description of the property to help API consumers better understand the requirements
#E We use a JSON pointer to reuse the OrderItem schema we have already defined

Alternatively, we can inherit the definition of the Order schema and extend it. In OpenAPI, we can inherit and extend a schema using the method of **model composition**. Model composition allows us to list different sets of schemas for a single object definition. The special keyword `allOf` is used in these cases to indicate that the object requires the combination of all the schemas listed. Listing 5.6 shows an alternative definition of the GetOrder schema using the allOf keyword.

5.6 Alternative implementation of the GetOrder schema using the allOf keyword

```
components:
  schemas:
    OrderItem:
      ...

    GetOrder:
      allOf:    #A
        - $ref: '#/components/schemas/Order'    #B
        - type: object    #C
          properties:
            status:
              type: string
              enum:
                - active
                - completed
                - cancelled
            created:
              type: integer
              description: Date in the form of UNIX timestmap
```

#A We use the allOf keyword to inherit all the properties already defined in one or more schemas within the same API specification. allOf is an array of objects and pointers
#B We use a JSON pointer to refer to the schema whose properties we want to inherit
#C We define a new object to include properties which are specific to GetOrder

In this case, the `GetOrder` object has a schema which results from the composition of two other schemas: the `Order` schema, and an unnamed schema defined within this object consisting of two keys: `status` and `created`. You can choose any of these methods to reuse your schemas. Model composition results in a cleaner and more succinct syntax, but it only works if the schemas are strictly compatible. If we decide to extend the schema for the `Order` model with new attributes which are specific for a POST request, then this schema won't be transferable to the `GetOrder` model anymore. In that sense, it's sometimes better to look for common elements among different schemas, and refactor their definitions out into an independent model that we can refer from other schemas using JSON pointers, as in the first definition of the `GetOrder` model. Listing 5.7 shows the specification for the GET `/orders/{order_id}` endpoint.

Listing 5.7 Specification for the GET /orders/{order_id} endpoint

```
paths:    #A
  /orders:
    get:
        ...    #B

  /orders/{order_id}:    #C
    parameters:    D
      - in: path
        name: order_id
        required: true
        schema:
          type: string
          format: uuid
```

```
  put:
    ...    #E

  get:    #F
    summary: Returns the details of a specific order    #G
    responses:    #H
      '200':    #I
        description: OK    #J
        content:    #K
          application/json:
            schema:
              $ref: '#/components/schemas/GetOrder'    #L
```

#A The specification for the GET /orders/{order_id} endpoint goes under the paths section of the API, together with all other endpoints we previously defined
#B We omit the schemas of the endpoints we previously defined to focus on the new changes
#C The new GET endpoint extends the definition of the /orders/{order_id} URL path
#D The definition of the URL path parameter order_id applies to all endpoints
#E We omit the schema for the HTTP method PUT since we've already seen it before
#F The schema for the new GET endpoint
#G We provide a summary description of the capabilities exposed by this endpoint
#H We define all the responses that can be returned by this endpoint under the responses keyword
#I Each response is an object where the key is the status code of the response and the values are the properties that describe the response
#J A brief description of the response. In this case, we describe it using the standard reason phrase[4] of a 200 status code (OK)
#K Description of the content types of the response and their corresponding payloads
#L Pointer to the schema that defines the payload in the response

As you can see from this definition, within the responses attribute of an endpoint we can describe all the responses which are relevant to the endpoint. In this case, we are only providing the specification for the 200 (OK) successful response, but we could also provide specific implementations of other status codes, such as error responses. As we mentioned in section 4.3.3, error responses are more generic, so we can use the components section of the API specification to provide generic definitions of those responses and then reuse them in the specification of our endpoints. Listing 5.8 shows a generic definition for a 404 response.

Listing 5.8 Generic 404 status code response definition

```
components:
  responses:    #A
    NotFound:    #B
      description: The specified resource was not found.    #C
      content:    #D
        application/json:
          schema:
            $ref: '#/components/schemas/Error'    #E

  schemas:    #F
    OrderItem:
```

[4] Reason phrase is the name given to the phrase usually associated with a status code, such as OK in 200, Created in 201, or No Content in 204. See R. Fielding, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing" (https://tools.ietf.org/html/rfc7230#section-3.1.2).

```
    ...    #G

  Error:    #H
    type: object
    properties:
      error:
        type: string
```

#A Generic response definitions go under the responses header of the API specification's component section
#B We name the response so that we can refer to it from other parts of the document using JSON pointers
#C We can provide a description to further clarify the purpose of this response
#D We define the content and payload of an error response just like we do for the standard response of an endpoint
#E We point to the schema that defines the payload of the error response
#F The schema definition of an error response goes under the schemas header of the API specification's component section, together with all other schemas we've seen earlier
#G We omit the definition of schemas we've seen before
#H Definition of a standard error schema

This specification for the 404 response can be reused in the specification of all our endpoints under the `/orders/{order_id}` path, since all of those endpoints are specifically designed to target a specific resource[5]. Listing 5.9 illustrates how we ca hook the 404 common response we defined previously under the GET /orders/{order_id} endpoint.

Listing 5.9 Using the common 404 response schema under the GET /orders/{order_id} endpoint

```
paths:
  ...    #A

  /orders/{orderId}:
    get:
      summary: Returns the details of a specific order
      parameters:
        - in: path
          name: orderId
          required: true
          schema:
            type: string
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/GetOrder'
        '404':    #B
           $ref: '#/components/responses/NotFound'    #C
```

#A We omit previous endpoint definitions to focus on the new changes
#B The new error response is for a 404 status code

[5] You may be wondering, if these are common responses to all the endpoints of a URL path, why can't we define the responses directly under the URL path and avoid repetition? The answer is this is not possible at the moment. The responses keyword is not allowed directly under a URL path, and therefore all responses relevant for a certain endpoint have to be defined directly under the endpoint. There's a long-running ticket request open in the OpenAPI repository to allow including common responses directly under the URL path, but it hasn't been accepted or rejected yet (https://github.com/OAI/OpenAPI-Specification/issues/521).

#C Pointer to the definition of the response schema

The specification for the `GET /orders` endpoint would use the same `GetOrder` schema that we used in `GET /orders/{order_id}`, but wrapped within an array. Listing 5.10 shows the implementation for the GET /orders endpoint.

Listing 5.10 Specification for the GET /orders endpoint

```
paths:
  /orders:
    get:    #A
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array    #B
                items:
                  $ref: '#/components/schemas/GetOrder'    #C

    post:
      ...    #D

  /orders/{order_id}:
    parameters:
      ...
```

#A The definition of the new GET /orders endpoint
#B The schema for the response payload is an array of orders
#C Each item in the array is defined by the GetOrder schema
#D We omit the definitions of other endpoints we have already seen before

This completes our journey for specifying our API's endpoints. You can use many more attributes within the definitions of your endpoints, such as `operationId`, `tags`, or `externalDocs`. These attributes are not strictly necessary for the specification of your endpoints, but they can be helpful in some cases to provide more structure to your API or to make it easier to group and refer to the endpoints. For example, you can use the `tags` attribute to create groups of endpoints which share some properties in common, while you can use the `operationId` attribute to create an easy-to-use identifier for each endpoint. For a full specification of the orders API, and some examples of how to use some of these additional attributes, see appendix A.

5.2.3 Defining the authentication scheme of the API

This sections explains how we can provide specifications for our API's security schemes. If our API is protected by a security scheme, the OpenAPI specification for the API must contain a description of such scheme. The specification of our security scheme tells consumers of our API how they must authenticate in order to send valid requests to our protected endpoints. The security definitions of the API go within the components section, under the header of `securitySchemes`.

OpenAPI allows us to define specifications for different security schemes, such as HTTP-based authentication, key-based authentication, OAuth2, and OpenID Connect. For a brief overview of these security schemes and how you document them in an OpenAPI specification, see appendix C. To illustrate how these specifications work, we'll use the example of JSON Web Token (JWT)-based authentication, which is the authentication mechanism that we implement for our API in chapter 10. A JWT is an encoded payload which contains claims about the user sending the request, and is usually carried in the Authorization HTTP Header Field. For more information on this topic, please refer to chapter 10. Listing 5.11 shows the defition of a security scheme based on JWT tokens. In OpenAPI, JWT-based authentication is a type of of HTTP-based bearer authentication.

Listing 5.11 Specification for a JWT-based security scheme

```
components:
  responses:
    ...    #A

  schemas:
    ...

  securitySchemes:    #B
    bearerAuth:    #C
      type: http    #D
      scheme: bearer    #E
      bearerFormat: JWT    #F
```

#A We omit the definitions of other elements we've included before in the components section of the API specification, such as responses and schemas
#B The security schemes used in the API are described under the securitySchemes header of the API's components section
#C A name for the security scheme we are defining here (it can be any name)
#D The type of security scheme we are describing. Other options are apiKey, oauth2, and openIdConnect
#E The scheme of security scheme we are describing. HTTP-based authentication allows basic and bearer schemes
#F The format of the security token

We name the model representing our security scheme `bearerAuth`, although we could choose any other name, as this is just the identifier for our model. The two required attributes for all types of security schemes in OpenAPI are `type` and `scheme`. In addition to these attributes, the bearer authentication scheme also requires that we specify the format of the authorization token, which in this case is JWT.

5.3 Generating the API specification with code

In this section, we see how we can generate API specifications from code using Python. In the previous sections of these chapter, we've been generating elements of the API specification for the orders service using YAML. That's a perfectly valid approach, and in practice, it's probably the most common approach to generating API specifications, since everybody understands YAML and any language can parse it. However, it wouldn't be an understatement to say that most people find it tedious to work with YAML. And as we have seen through this chapter, API specifications in YAML can get a tad long. Long documents are

more difficult to read and maintain. We can make our life easier by using libraries that help us produce API specifications using a few lines of code.

We are going to see how to generate an API specification using the Python library apispec. You can find the full specification for the orders API using apispec under the folder ch04 of the repository provided with this book. apispec is a Python library that comes from the marshmallow ecosystem, and it allow us to generate specifications using a variety of plugins. In this section, we use marshmallow plugin to define our API schemas. Marshmallow is one of the most popular data validation libraries in Python, and it contains a very expressive syntax that allows us to declare with precision the requirements of our schemas.

CREATING A VIRTUAL ENVIRONMENT AND INSTALLING DEPENDENCIES

Before we get started with the code, let's make sure we create a virtual environment for this task, and we install the dependencies that we'll need to work on this implementation. Listing 5.12 shows you how to create a virtual environment and install the necessary dependencies.

Listing 5.12 Create a virtual environment and install the required dependencies

```
$ pipenv -three    #A

$ pipenv shell    #B

$ pipenv install apispec marshmallow    #C

#A Create the virtual environment with pipenv
#B Activate the virutal environment
#C Install the dependencies
```

INITIALIZING THE API SPECIFICATION OBJECT

Now that our virtual environment is ready, we can start coding! Let's first create an instance of the `APISpec` class of the apispec library, which represents the container for the whole specification. We'll register our schemas and paths with this object, and APISpec will know how to put these together to produce a valid OpenAPI 3.0.2 specification. Create a file called `oas.py` and add to it the code shown in listing 5.13 to initialize the APISpec class, which will represent our API specification object.

Listing 5.13 Initialization of the APISpec class

```
from apispec import APISpec    #A
from apispec.ext.marshmallow import MarshmallowPlugin    #B


spec = APISpec(
    title='Orders API',    #C
    version='1.0.0',    #D
    openapi_version='3.0.2',    #E
    plugins=[MarshmallowPlugin()],    #F
    **{'info': {'description': 'API that allows you to manage orders for CoffeeMesh'}},
       #G
)
```

#A We import the APISpec class from apispec. The instance of this class will represent our API specification object

#B We'll use the MarshmallowPlugin to be able to define schemas using Marshmallow
#C We set the title of the API specification on initialization
#D We specify the version of the orders API
#E We specify the version of OpenAPI that we want to use to generate the specification
#F We specify the list of plugins that we want to use
#G We pass a dictionary with additional elements of the API specification that we want to specify but are not parametrized in the APISpec initializer

The `APISpec` class accepts parameters for the title of the API specification, its version, the version of OpenAPI that we want to produce, and a list of plugins that we wish to use to generate the specification. In this case, we'll use the `MarshmallowPlugin` to help us define our schemas. In addition to these parameters, it's also possible to pass a dictionary of additional values that we want to include in the specification and are not explicitly declared by `APISpec`. Such values must be declared at the specific level within the document where we want to include them, providing the full path to the property. In the above listing, we declare that we want to include a `description` property within the `info` property of the specification.

IMPLEMENTING THE SCHEMA DEFINITIONS FOR THE ORDERS API

Once we have got an instance of the `APISpec` class, we can start defining schemas and paths and registering them with the instance. In this section, we'll learn how we can use marshmallow to implement the schema definitions of the orders API: `GetOrderSchema`, `CreateOrderSchema`, and `OrderItemSchema`. Listing 5.14 provides implementations for these schema definitions.

Listing 5.14 Schema definitions for OrderItemSchema, CreateOrderSchema, and GetOrderSchema

```
from apispec import APISpec
from apispec.ext.marshmallow import MarshmallowPlugin
from marshmallow import Schema, fields, validate    #A


specification = APISpec(
    title='Orders API',
    version='1.0.0',
    openapi_version='3.0.2',
    plugins=[MarshmallowPlugin()],
    **{'info': {'description': 'API that allows you to manage orders for CoffeeMesh'}}
)


class OrderItemSchema(Schema):    #B
    product = fields.String(required=True)    #C
    size = fields.String(
        required=True, validate=validate.OneOf(['small', 'medium', 'big'])    #D
    )
    quantity = fields.Integer(
        default=1, validate=validate.Range(1, min_inclusive=True)    #E
    )


class CreateOrderSchema(Schema):
```

```
    order = fields.List(fields.Nested(OrderItemSchema), required=True)    #F


class GetOrderSchema(Schema):
    id = fields.UUID(required=True)    #G
    created = fields.Integer(required=True)
    status = fields.String(
        required=True, validate=validate.OneOf(['active', 'cancelled', 'completed'])
    )
    order = fields.List(fields.Nested(OrderItemSchema), required=True)


specification.components.schema('GetOrderSchema', schema=GetOrderSchema)    #H
specification.components.schema('CreateOrderSchema', schema=GetOrderSchema)
```

#A We import classes and functions from marshmallow that we'll need to implement our schema definitions
#B Each schema definition is represented by a class which inherits from the Schema class from marshmallow
#C Every property of the schema is implemented as an instance of any of the classes in marshmallow's field module
#D We can add validation rules by using any of the validation classes from marshmallow's validation module. In this case, we use the OneOf class to represent an enum in JSON Schema
#E We can also use the validate.Range class to specify the minimum and maximum values of an integer
#F To describe a JSON Schema array with marshmallow, we use the fields.List class in combination with fields.Nested to specify the schemas of the items in the array
#G In JSON Schema, a UUID is not a type, but a string with UUID format. In marshmallow, we can use the fields.UUID class to describe such specification
#H We register the the schemas with the instance of APISpec using APISpec the schema method of APISpec's components attribute, which is itself an instance of apispec.core's Components class

Each property of a schema is defined as a field in Marshmallow, and the library provides a rich interface of field types that we can use to build our specification. By default, properties are considered to be not required, so when a property is required, we need to set the `required` parameter of a field to `True`. For highly specific validations, marshmallow provides the `validate` module with a rich collection of validation classes. In the above listing, we use `Range` class to define the range of values accepted for the `quantity` property of the `OrderItemSchema` model, and we use the `OneOf` to define the list of accepted values that the `status` property of the `GetOrderSchema` and the `size` property of the `OrderItemSchema` can take. In order to register the schemas with our specification object, we simply use the `schema` method of the `Components` class, which is available `APISpec.components` attribute. You can see what the API looks like at this point by adding the following print statement at the end of the file: `print(specification.to_yaml())`.

DOCUMENTING THE API ENDPOINTS USING APISPEC

Now that we have seen how to define our specification schemas, let's see how we can define the URL paths of the API. apispec offers different possibilities to define URL paths. At the most basic level, URL paths can be defined using dictionaries. Listing 5.15 illustrates how the GET /orders endpoint can be documented using this approach.

Listing 5.15 Implementation of the GET /orders endpoint specification using APISpec's path method

```
specification.path(    #A
    path='/orders',    #B
    operations={    #C
        'get': {    #D
            'responses': {    #E
                '200': {
                    'content': {
                        'application/json': {
                            'schema': 'GetOrderSchema'
                        }
                    }
                }
            }
        }
    }
)
```

#A We can use the APISpec.path method to describe the endpoints in our API
#B We specify the URL path using the path parameter
#C The HTTP methods supported by the URL path are specified within the operations parameter
#D Definition of the HTTP method GET for the /orders URL path
#E Specification of the responses for the GET /orders endpoint. At this point, the rest of the implementation looks very similar to an OpenAPI specification document

This is OK, but we are basically replicating YAML or JSON in Python, and we are not leveraging the most important advantage of using code to build our API, which is being able to employ reusable components. Let's create a few functions that allow us to define the successful response of a method, the request payload, and the parameters of a URL by passing a few parameters. Create a new file called `oas_helpers.py` and include in it the functions implemented in listing 5.16.

Listing 5.16 Functions to help us produce API specifications with less code using apispec

```
def make_response(schema=None, status_code='200', content_type='application/json',
       description=None):    #A
    response = {
        status_code: {
            'content': {
                content_type: {
                    'schema': schema
                }
            },
        }
    }
    if description is not None:    #B
        response[status_code]['description'] = description
    return response


def make_request_body(schema, required=True, content_type='application/json'):    #C
    return {
        'required': required,
```

```
        'content': {
            content_type: {
                'schema': schema
            }
        }
    }


def make_parameter(in_, name, schema, required=True, description=None):    #D
    return {
        'in': in_,
        'name': name,
        'schema': schema,
        'required': required
    }
```

#A Helper function to build response specification for endpoints. Each of the parameters in this function has default values which reflect the most common use cases for the orders API: the most frequently used status code in the responses is 200, and the content type is always application/json. The schema of the response is specific to each endpoint, so we can't provide a default schema, and instead set it to None to account for the case of empty response represented by 204 responses. We also make it optional to include a description of the response
#B If a description is provided, we include it in the response
#C Helper function to build request payload specifications for endpoints. Here also we provide default values for the function parameters that reflect the most common uses cases for the orders API
#D Helper function to build description of URL parameters. Again, we provide default values for the function parameters that reflect the most common use cases in the orders API

`make_response` allows us to generate a response object for an endpoint. Since 200 is the most common status code returned by our endpoints in case of success, we make that a default value of the response. Also, since our API uses JSON exclusively to represent data, we make application/json the default content data type of the responses. If a response contains a payload, such payload is represented by a schema. In this case, we set `schema` to `None` by default since not all endpoints return payloads (for example, DELETE endpoints normally don't need to return payloads). If the consumer of this function doesn't specify a schema, we understand that the response will not contain any payload.

`make_request` allows us to generate specifications for the request payload of the endpoints. This function is only necessary for endpoints that accept a payload in their request, and the most important argument in its signature is the `schema` argument. By default, we assume that the payload will be required and that the content type of the payload will be application/json. Finally, the `make_parameter` function allows us to generate the specification for the parameters accepted in an endpoint by parametrizing the properties that we can use to describe them. We can now import these functions into oas.py and use them to generate specifications for the orders API endpoints, as shown in listing 5.17.

Listing 5.17 Implementation of the orders API endpoints' specifications

```
from apispec import APISpec
from apispec.ext.marshmallow import MarshmallowPlugin
from marshmallow import Schema, fields, validate

from oas_helpers import make_response, make_request_body, make_parameter
```

```
specification = APISpec(
    title='Orders API',
    version='1.0.0',
    openapi_version='3.0.2',
    plugins=[MarshmallowPlugin()],
    **{'info': {'description': 'API that allows you to manage orders for CoffeeMesh'}}
)

...

specification.path(
    path='/orders',    #A
    operations={
        'get': {    #B
            'description': (    #C
                'A list of orders made by the customer sorted by date. Allows '
                'to filter orders by range of dates.\n'
            ),
            'responses': make_response(    #D
                {'type': 'object', 'properties': {'data': {'type': 'array', 'items':
      {'$ref': '#/components/schemas/GetOrderSchema'}}}},
                description='A JSON array of orders'
            )
        },
        'post': {    #E
            'summary': 'Creates an Order',    #F
            'requestBody': make_request_body('CreateOrderSchema'),    #G
            'responses': make_response(    #H
                'GetOrderSchema', status_code='201',
                description='A JSON representation of the created order'
            )
        }
    }
)


specification.path(
    path='/orders/{order_id}',    #I
    parameters=[make_parameter(in_='path', name='order_id', schema={'type':
      'string'})],    #J
    operations={
        'get': {    #K
            'summary': 'Returns the details of a specific order',
            'responses': make_response(
                'GetOrderSchema', description='A JSON representation of an order'
            )
        },
        'put': {    #L
            'description': 'Replaces an existing order',
            'requestBody': make_request_body('CreateOrderSchema'),
            'responses': make_response(
                'GetOrderSchema', description='A JSON representation of an order'
            )
        },
        'delete': {    #M
            'description': 'Deletes an existing order',
            'responses': {'204': {'description': 'The resource was deleted
      successfully'}}
        }
    }
```

```
)

print(specification.to_yaml())     #N
```

#A Specification of the endpoints under the /orders URL path
#B Specification of the HTTP method GET under the /orders URL path
#C A description to help API consumers understand the behavior of the GET /orders endpoint
#D We use the make_response helper to build the specification for this endpoint's response
#E Specification for the HTTP method POST under the /orders URL path
#F We provide a summary description to help API consumers understand the behavior of the POST /orders endpoint
#G We use the make_request_body helper function to build the specification for the request payload in this endpoint
#H We use the make_response helper function to build the specification for this endpoint's response. In this case, we override the default value for status_code and we provide a description
#I Specification of the endpoints under the /orders/{order_id} URL path
#J We use the make_parameter helper function to build the specification for the URL path parameter in this URL path
#K Specification of the HTTP method GET under the /orders/{order_id} URL path
#L Specification of the HTTP method PUT under the /orders/{order_id} URL path
#M Specification of the HTTP method DELETE under the /orders/{order_id} URL path
#N We generate the API specification in YAML format and print it to the terminal so that we can verify that it looks as intended

You can compare the specification generated with this approach with the specification that we wrote manually over the course of this chapter, and you'll be pleased to see that both documents are nearly identical. The three helper functions that we implemented to help generating the components of our specification are just the tip of the iceberg of what you can do when you use code to generate API specification, and hopefully they give you some inspiration to work on more and better helper components that give you additional capabilities to represent your paths and endpoints.

5.4 Summary

- JSON Schema is a specification format that we can use to define the structure and format of a JSON document, and therefore we can use it to define the payloads of our API requests and responses
- JSON Path is a syntax that allows us to point to a specific attribute in a JSON document
- A JSON Pointer is a specific syntax that we can use to refer to a specific object in a JSON Schema, using JSON Path syntax to point to the object. We can use JSON Pointers in order refactor an API specification by creating generic objects, resulting in a more readable and maintainable specification
- A Swagger/OpenAPI specification contains the following sections:
    - Metadata: contains information about the specification itself, such as the version of OpenAPI used to produce the specification
    - Servers: contains information about the URLs for the API, such as the URL for the production API
    - Paths: describes the endpoints exposed by the API, including the schemas for the API requests and responses, and any relevant URL query strings
    - Components: describes objects and payloads used across the API

    - Authentication: describes the authentication and authorization methods required in the API

- You don't need to write API specifications manually. You can use different libraries available in Python to generate API specifications from code. In addition, you can write your own helper components to make it easier to produce such specifications.

6

Implementing REST APIs with Python

This chapter covers

- Adding URL query parameters to an endpoint using FastAPI
- Disallowing the presence of unknown properties in a payload using Pydantic and marshmallow
- Automatically generating Pydantic models from an OpenAPI specification
- Implementing a REST API using Flask-smorest
- Defining validation schemas and URL query parameters using marshmallow

In this chapter, we implement two REST APIs from the CoffeeMesh platform, the on-demand coffee delivery application that we introduced in chapter 1. We'll implement the APIs for the orders service and for the kitchen service. The orders service is the main gateway to CoffeeMesh for customers of the platform. Through it they can place orders, pay for them, update them, and keep track of them. The kitchen service takes care of scheduling orders for production in the CoffeeMesh factories and keeps track of their progress. We'll learn best practices for the implementation of REST APIs as we work through these examples.

In chapter 2 we implemented part of the orders API. In the first sections of this chapter, we pick up the orders API where we left it in chapter 2 and implement its remaining features using FastAPI. FastAPI is a highly performant web framework for Python, and a very popular choice for implement REST APIs. We'll learn how to add URL query parameters to our endpoints using FastAPI. As we saw in chapter 2, FastAPI uses Pydantic for data validation, and in this chapter we'll see how we can instruct Pydantic to forbid the presence of unknown fields in a payload. We'll learn about the tolerant reader pattern and balance its benefits against the risk of API integration failures due to errors such as typos.

After completing the implementation of the orders API, we'll implement the API for the kitchen service. The kitchen service schedules orders for production in the factory and keeps track of their progress. We'll implement the kitchen API using Flask-smorest, a popular API framework built on

top of Flask and marshmallow. We'll learn to implement our APIs following Flask application patterns, and we'll see how we define validation schemas using marshmallow.

By the end of this chapter, you'll have a thorough understanding of how to implement REST APIs in Python using two of its most popular libraries for this type of job. You'll see how the principles for implementing REST APIs transcend the implementation details of each framework and can be applied regardless of the technology that you use. The code for this chapter is available under folder ch06 in the repository provided with this book. Folder ch06 contains two subfolders: one folder for the orders API (ch06/orders) and one folder for the kitchen API (ch06/kitchen).

With that said, and without further ado, let's get cracking!

6.1 Overview of the Orders API

In this section, we'll recap the minimal implementation of the orders API that we undertook in chapter 2. The full specification of the orders API can be found in the GitHub repository provided with this book, under `ch06/orders/oas.yaml`. Before we jump directly onto the implementation, let's briefly analyze the specification and see what's left to implement.

In chapter 2, we implemented the API endpoints of the orders API, and we provided schema definitions to validate request and response payloads. We intentionally skipped implementing the service or business layer of the application, as that's a complex task that we'll tackle in chapter 7.

As a reminder, the endpoints exposed by the orders API are the following:

- `/orders`: allows us to retrieve lists (GET) of orders and to create orders (POST).
- `/orders/{order_id}`: allows us to retrieve the details of a specific order (GET), to update and order (PUT), and to delete an order (DELETE).
- `/orders/{order_id}/cancel`: allows us to cancel an order (POST).
- `/orders/{order_id}/pay`: allows us to pay for an order (POST).

POST /orders and PUT `/orders/{order_id}` require request payloads that define the properties of an order, and in chapter 2 we implemented schemas for those payloads. What's missing from the implementation is the URL query parameters for the GET /orders endpoint. Also, the schemas that we implemented don't validate for the presence of illegal properties in the payloads. As we'll see in section 6.3, this is fine in some situations, but it may lead to integration issues in some cases, and we'll see how we can configure the schemas to invalidate payloads with illegal properties. Finally, we'll have a look at the datamodel-code-generator library, which allows us to automatically generate the code for the validation schemas directly from the specification.

If you want to follow along with the code that we add in this chapter on top of what we wrote in chapter 2, create a folder called orders and copy within it the code from ch02/orders. Remember to install the dependencies and activate the virtual environment running the following commands:

```
$ pipenv install
$ pipenv shell
```

You can start the server by running the following command:

```
$ uvicorn orders.app:app --reload
```

FASTAPI + UVICORN REFRESHER We implement the orders API using the FastAPI framework, a popular Python framework for building REST APIs. FastAPI is built on top of Starlette, an asynchronous web server implementation. To execute our FastAPI application, we use uvicorn, another asynchronous server implementation that handles efficiently incoming requests to the server.

The `--reload` flag makes uvicorn watch for changes on your files, so that any time you make a change, the application is reloaded with the latest changes. This saves you the time of having to stop the server and start it again every time you make changes to the code.

With this covered, let's now complete the implementation of the orders API!

6.2 URL query parameters for the Orders API

In this section, we enhance the GET /orders endpoint of the orders API by adding URL query parameters. We'll also implement validation schemas for the parameters. In section 4.5, we learned that URL query parameters allow us to filter the results returned by an endpoint. In section 5.2.2, we established that the GET /orders endpoint accepts URL query parameters to filter orders by whether they're cancelled, and also to limit the list of orders returned by the endpoint. The specification for these parameters is shown in listing 6.1.

Listing 6.1 Specification for the GET /orders URL query parameters

```
paths:
  /orders:
    get:
      parameters:
        - name: cancelled
          in: query
          required: false
          schema:
            type: boolean
        - name: limit
          in: query
          required: false
          schema:
            type: integer
```

We need to implement two URL query parameters: cancelled (Boolean) and limit (integer). None of them are required, so users should be able to call the GET /orders endpoint without specifying them at all. Let's see how we do that.

Implementing URL query parameters for an endpoint is easy with FastAPI. All we need to do is include them in the function signature for the endpoint, and use type hints to create validation rules for them. Since the query parameters are optional, we'll mark them as such using the `Optional` type and we'll set their default values to `None`. Listing 6.2 shows how to implement the URL query parameters from the specification shown in Listing 6.1 for the GET /orders endpoint.

Listing 6.2 Implementation of URL query parameters for GET /orders

```
@app.get('/orders', response_model=List[GetOrderSchema])
def get_orders(cancelled: Optional[bool] = None, limit: Optional[int] = None):      # A
    ...
```

#A To add URL query parameters in FastAPI, we simply need to include them in the function signature. Because these URL parameters are optional, we mark them with the Optional type and set their default values to None

Now that we have query parameters available in the GET /orders endpoint, how should we handle them within the function? Since the query parameters are optional, we'll first check whether they've been set by the user. We can do that by checking whether their values are other than None. If the values have been set, we'll use them to filter the list of orders. Listing 6.2 shows we can handle URL query parameters within the function body of the GET /orders endpoint. Study figure 6.1 to understand how the decision flow to filter the list of orders based on the query parameters works.

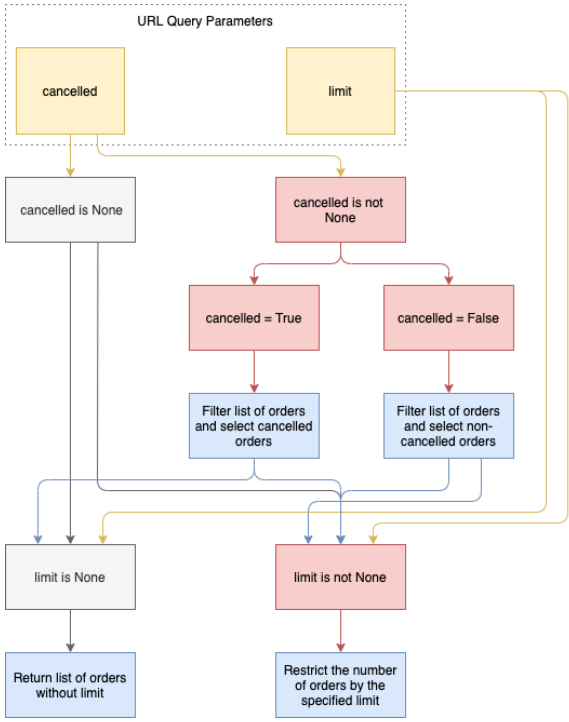


Figure 6.1 Ordering filtering flow based on query parameters. If the cancelled parameter is set to True or False, we filter the list of orders by whether they're cancelled or not. After this step, we check whether the limit parameter is set. If limit is set, we only return the corresponding number of orders from the list.

Listing 6.2 Implementation of URL query parameters for GET /orders

```
@app.get('/orders', response_model=List[GetOrderSchema])
def get_orders(cancelled: Optional[bool] = None, limit: Optional[int] = None):      # A
    if cancelled is None and limit is None:    #A
        return orders
```

```
    query_set = [order for order in orders]    #B

    if cancelled is not None:    #C
        if cancelled:
            query_set = [order for order in query_set if order['status'] == 'cancelled']
        else:
            query_set = [order for order in query_set if order['status'] != 'cancelled']

    if limit is not None and len(query_set) > limit:    #D
        return query_set[:limit]

    return query_set
```

#A If the parameters haven't been set, we return immediately
#B If any of the parameters has been set, we'll build a filtered list. To do that, we make a copy of the orders list into a variable named query_set
#C Because the default value of cancelled is None, we need to check whether it's value is not None. Just checking if not cancelled doesn't make the cut since cancelled is a Boolean and its value can be False. Make sure you never fall for that common trap!
#D If the parameter limit has been set and its value is lower than the length of the query_set, we'll use limit to return a subset of the query_set

Now that we know how to add URL query parameters to our endpoints, let's see how we can enhance the validation mechanisms of our schemas. Move on to the next section to learn more about that!

6.3 Validating payloads with unknown fields

Until now, our Pydantic models have been tolerant with the request payloads. If an API client sends a payload with fields which haven't been declared in our schemas, the payload will be accepted. As we'll see in this section, this may be convenient in some cases, but misleading or dangerous in other contexts. To handle the latter case, in this section we learn how to configure Pydantic to forbid the presence of unknown fields in a payload. Unknown fields are fields that haven't been defined as part of the schema for that payload.

> **PYDANTIC REFRESHER** As we saw in chapter 2, FastAPI uses Pydantic to define validation models for our APIs. Pydantic is a popular data validation library for Python which offers a modern interface that allows you to define data validation rules using type hints.

In chapter 2 we implemented the schema definitions of the Orders API following the tolerant reader pattern[1]. The tolerant reader pattern follows Postel's law, which recommends to

> *be conservative in what you do, be liberal in what you accept from others.*[2]

In the field of web APIs, this means that the payloads that we send to the client should be properly validated and compliant with the specification, while the payloads that we accept in the server should allow for typos and for the presence of unknown fields. JSON Schema follows this

[1] Martin Fowler, "TolerantReader" (https://martinfowler.com/bliki/TolerantReader.html [accessed 1st November 2020]).
[2] Jon Postel, ed., "Transmission Control Protocol", Internet Engineering Task Force, RFC 761, p. 13 (https://tools.ietf.org/html/rfc761 [accessed 1st November 2020]).

pattern by default, and unless explicitly declared, a JSON Schema object accepts any kind of property, and validation only ensures that none of the required properties are missing.

This pattern is useful in situations where the API is not fully consolidated or likely to change frequently, and we want to be able to make changes to it without breaking integrations with existing clients. However, in some other cases, like we saw in chapter 2, the tolerant reader pattern can introduce new bugs or lead to unexpected integration issues.

For example, `OrderItemSchema` has three properties: product, size, and quantity. Product and size are required properties, but quantity is optional, and if missing, the server assigns to it the default value of 1. In some scenarios, this can lead to confusing situations. Imagine a client sent a payload setting the quantity property, but making a typo in its representation, for example with the following payload:

```
{
  "order": [
    {
      "product": "capuccino",
      "size": "small",
      "quantit": 5
    }
  ]
}
```

Using the tolerant reader implementation, we ignore the field "quantit" from the payload and assume that the quantity property is missing and will set its value to the default of 1. This situation can be confusing for the client, who intended to set a different value for quantity.

> **THE API CLIENT SHOULD'VE TESTED THEIR CODE!** You can argue that the client should've tested their code and verified that it works properly before calling the server. And you're right. But in real life, code goes untested, or is not properly tested for many reasons, and a little bit of extra validation in the server will help in those situations. If we check the payload for the presence of illegal properties, this error will be caught and reported to the client.

How can we accomplish this using Pydantic? Pydantic provides a rich interface that helps us customize and configure the behavior of our schemas. To disallow additional attributes, we need to define a Config class within our models, and set the `extra` property to `forbid`. Listing 6.2 illustrates the changes that we need to make to our schemas to enable this behavior.

Listing 6.2 Disallow additional properties in models

```
class OrderItemSchema(BaseModel):
    product: str
    size: SizeEnum
    quantity: int = Field(default=1, ge=1, example=1)

    class Config:    #A
        extra = Extra.forbid


class CreateOrderSchema(BaseModel):
    order: List[OrderItemSchema]

    class Config:
```

```
        extra = Extra.forbid


class GetOrderSchema(CreateOrderSchema):    #B
    id: UUID
    created: int = Field(description='Date in the form of UNIX timestmap')
    status: StatusEnum
```

#A We can customize the behavior of our Pydantic models by adding a nested class named Config. In this case, we use the Config class to ban from the payload the presence of any properties that haven't been defined in the schema

Let's test this new functionality. Run the following command to start the server:

```
$ uvicorn orders.app:app
```

As we saw in chapter 2, FastAPI is capable of dynamically generating documentation from the implementation, and is also able to render an interactive visualization of the API (a Swagger UI), which we can use to easily test the endpoints. We'll use this UI to test our new validation rules with the following payload:

```
{
  "order": [
    {
      "product": "string",
      "size": "small",
      "quantit": 5
    }
  ]
}
```

> **WHAT IS A SWAGGER UI?** As we mentioned in chapter 2, a Swagger UI is a popular style for representing interactive displays of REST APIs. They provide a user-friendly interface that helps us understand the API implementation and test its endpoints. Another popular UI for REST interfaces is Redoc (https://github.com/Redocly/redoc).

To get to the Swagger UI, visit the following address in a browser: http://127.0.0.1:8000/docs, and follow the steps in figure 6.2 to learn how to execute a test against the POST /orders endpoint.

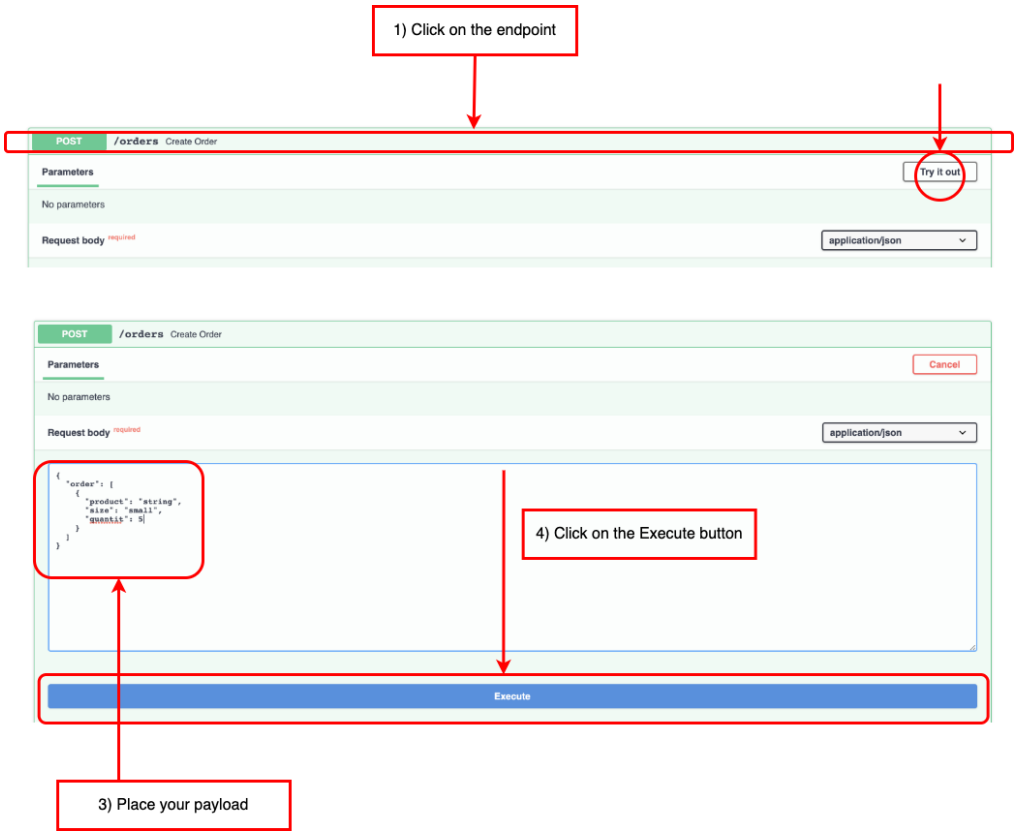


Figure 6.2 Testing the API through Swagger UI: to test an endpoint, click on the endpoint itself, then click on the Try it out button, then click on the Execute button.

After running this test, you'll see that now the FastAPI invalidates this payload and returns a helpful 422 response with the following message: "extra fields not permitted."

Now that we know how to implement and configure our models using Pydantic, let's this one step further and see how we can automatically generate Pydantic models from an OpenAPI specification!

6.4 Automatically loading validation schemas from the specification

In this section, we learn how to automatically load validation schemas from an OpenAPI specification using Pydantic. In section 2.4 we manually wrote the code for the schemas in the orders API specification. Doing this manually is fine when we have a few simple schemas, but, wouldn't it awesome if we had a tool that could generate the schemas automatically for us? There's such a tool! It's a library called datamodel-code-generator and it's very easy to use. Install the library by running the following command:

```
$ pipenv install datamodel-code-generator
```

You may run into dependency conflict issues. If that's the case, run pipenv with the `--skip-lock` flag, which instructs pipenv to ignore the Pipfile.lock file and install directly from Pipfile. This allows pipenv to search for compatible versions of the dependencies:

```
$ pipenv install datamodel-code-generator --skip-lock
```

Now that the library is installed, we can use it to generate schemas from the specification. To do so, run the following command from the `ch06/orders` directory:

```
$ datamodel-codegen --input oas.yaml --output orders/api/auto_schemas.py
```

We're outputting the automatically generated schemas into a new file called `auto_schemas.py`, so that we can differentiate it from our own schemas in `schemas.py`. Listing 6.3 shows the code generated by datamodel-code-generator, with the major differences with our manually implemented schemas highlighted. As you can see, the models generated by datamodel-code-generator are nearly identical to the models we wrote manually in chapter 2. The main differences are the following:

- The automatically generated file contains some metadata at the top with the OpenAPI file that was used to generate the models and when it was done.
- It also imports `annotations` from the `__future__` module. This ensures that annotations or type hints are not evaluated at import time, which makes it possible to refer to annotations before they have been declared (for more information, see https://livebook.manning.com/book/classic-computer-science-problems-in-python/chapter-2/101 and https://www.python.org/dev/peps/pep-0563/).
- The automatically generated model for `GetOrderSchema` contains a slightly different implementation of the `created` property, adding an ellipsis as the first argument for the `Field` function[3]. Ellipses are used in Pydantic to make sure that optional properties are always present in the payload. In our case the ellipses are not needed since `created` is not an optional property, so you can delete them.
- datamodel-code-generator doesn't handle the `additionalProperties` of the OpenAPI specification, so you need to add manually the `Config` class to the models with the `extra` property set to `forbid`, as we showed in listing 6.2.
- In the automatically generated file, `GetOrderSchema` doesn't inherit from `CreateOrderSchema`. This is reasonable, since datamodel-code-generator is only trying to generate schemas that reflect as best as possible the definitions of the OpenAPI specification, while inheritance is a programming optimization to reuse existing code.

Listing 6.3 Automatically generated schemas

```
# generated by datamodel-codegen:    #A
#   filename:  oas.yaml
#   timestamp: 2020-11-07T11:14:21+00:00
```

[3] This is not a typo, Pydantic's `Field` is a function, never mind the capital `F`.

```
from __future__ import annotations    #B

from enum import Enum
from typing import List, Optional
from uuid import UUID

from pydantic import BaseModel, Field, conint


class Size(Enum):
    small = 'small'
    medium = 'medium'
    big = 'big'


class OrderItemSchema(BaseModel):
    product: str
    size: Size
    quantity: Optional[conint(ge=1)] = 1


class CreateOrderSchema(BaseModel):
    order: List[OrderItemSchema]


class Status(Enum):
    created = 'created'
    paid = 'paid'
    progress = 'progress'
    cancelled = 'cancelled'
    dispatched = 'dispatched'
    delivered = 'delivered'


class GetOrderSchema(BaseModel):    #C
    id: UUID
    created: int = Field(..., description='Date in the form of UNIX timestmap')    #C
    status: Status
    order: List[OrderItemSchema]
```

#A Metadata generated by the datamodel-code-generator library
#B Importing annotations from __future__ allows us to make references to a type before it's been declared
#C Unlike what we did in our model definitions, datamodel-code-generator doesn't make GetOrderSchema inherit from CreateOrderSchema, since the purpose of datamodel-code-generator is to reflect the API specification as accurately as possible, and not to figure out how to optimize code

When should I use datamodel-code-generator?

Should you use datamodel-code-generator or should you manually implement your schemas? If your OpenAPI file contains a few simple models, datamodel-code-generator may not save you a great deal of time, and as we have seen, you should double-check the automatically generated models to ensure they accurately reflect the specification. However, if your specification contains numerous and complex schemas, using datamodel-code-generator can help you save time by generating boilerplate code which only needs a few changes to reflect the requirements of the API specification.

This concludes our journey to implement the Orders API with FastAPI. It's now time to move on to implementing the API for the Kitchen service, for which we'll use a new stack: Flask + marshmallow. Let's get on with it!

6.5 Overview of the kitchen API

In this section, we get an overview of the implementation requirements for the API of the kitchen service. As you can see in figure 6.3, the kitchen service manages the production of orders made by customers. Customers interface with the kitchen service through the orders service when they place an order or check its status. CoffeeMesh staff can also use the kitchen service directly to check how many orders are scheduled for production and to manage them.

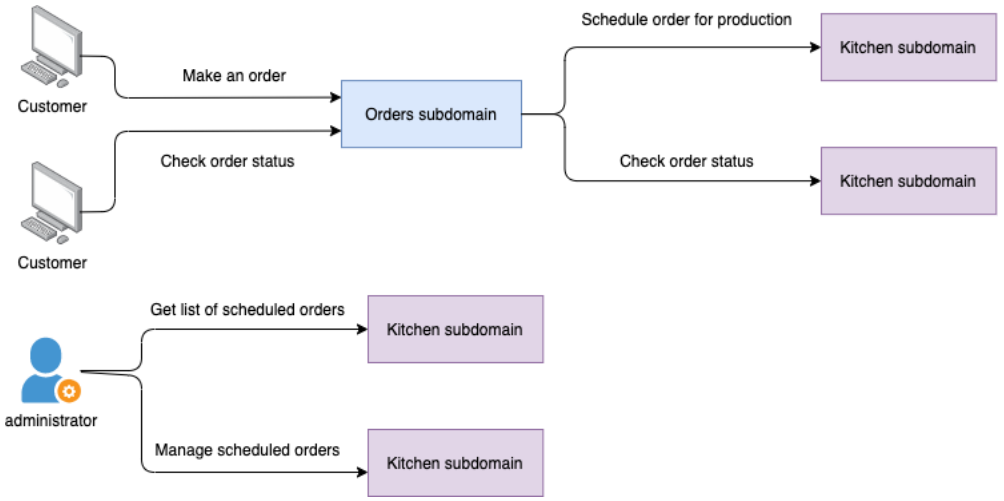


Figure 6.3 The kitchen service takes care of scheduling orders for production and it offers an interface which allows customers to keep track of their progress. CoffeeMesh staff members also use the kitchen service to manage scheduled orders.

The specification for the kitchen API is provided under `ch06/kitchen/oas.yaml` in the repository provided with this book. The kitchen API contains four URL paths (see figure 6.4 for additional clarification):

- `/kitchen/schedule`: allows us to schedule an order for production in the kitchen (POST) and to retrieve a list of orders scheduled for production (GET).
- `/kitchen/schedule/{schedule_id}`: allows us to retrieve the details or a scheduled order (GET), to update its details (PUT), and to delete it from our records (DELETE).
- `/kitchen/schedule/{schedule_id}/status`: allows us to retrieve the status of an order scheduled for production.
- `/kitchen/schedule/{schedule_id}/cancel`: allows us to cancel a scheduled order.

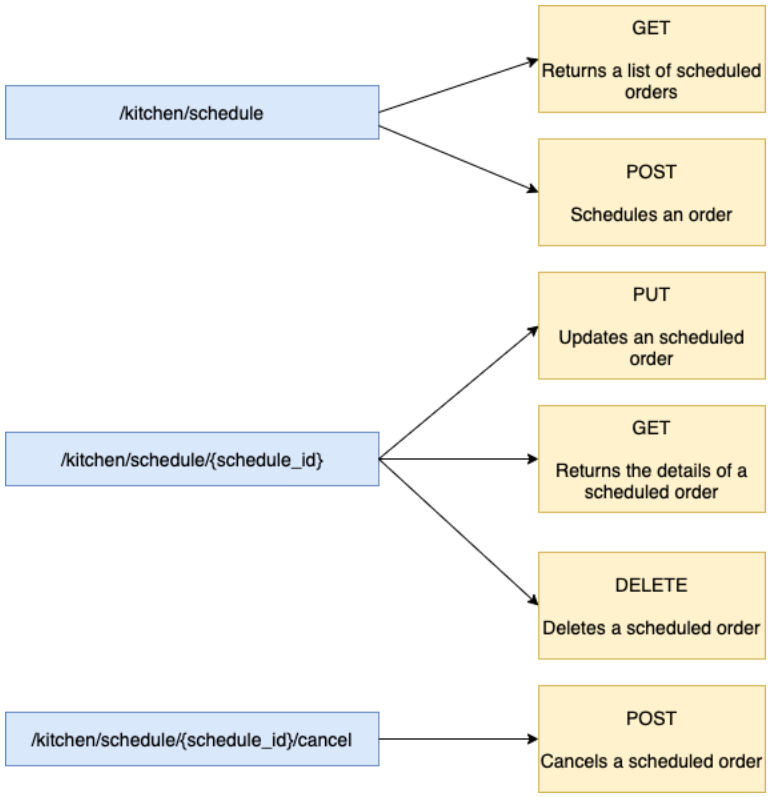


Figure 6.4: Endpoints of the kitchen API. The kitchen API has three URL paths: /kitchen/schedule, which exposes a GET and a POST endpoint; /kitchen/schedule/{schedule_id}, which exposes a PUT, a GET, and a DELETE endpoints; and kitchen/schedule/{schedule_id}/cancel, which exposes a POST endpoint.

The kitchen API contains three schemas: `OrderItemSchema`, `ScheduleOrderSchema`, and `GetScheduledOrderSchema`. The `ScheduleOrderSchema` represents the payload required to schedule an order for production, while the `GetScheduledOrderSchema` represents the details of an order that has been scheduled with the kitchen. Just like in the orders API, `OrderItemSchema` represents the details of each item in an order.

Like we did in chapter 2, we'll keep the implementation simple and focus only on the API layer or the kitchen service. We'll mock the business layer with an in-memory representation of the schedules managed by the service. In chapter 7, we'll learn service implementation patterns that will help us implement the business layer.

6.6 FastAPI vs flask-smorest

This section introduces the stack that we'll use to build the kitchen API and offers a comparison with FastAPI. To build the kitchen API, we'll use Flask-smorest. Flask-smorest is a REST API

framework built on top of Flask and marshmallow[4]. Flask is a popular lightweight framework for building web applications. Marshmallow is a popular data validation library which can handle conversion of complex data types to and from native Python objects. Flask-smorest builds on top of both frameworks, which means we implement our schemas using marshmallow, and we implement our API endpoints following the patterns of a typical Flask application, as illustrated in figure 6.5 (study this figure carefully and compare it with the figure we provided for FastAPI in the sidebar "What is FastAPI?" in section 2.3).

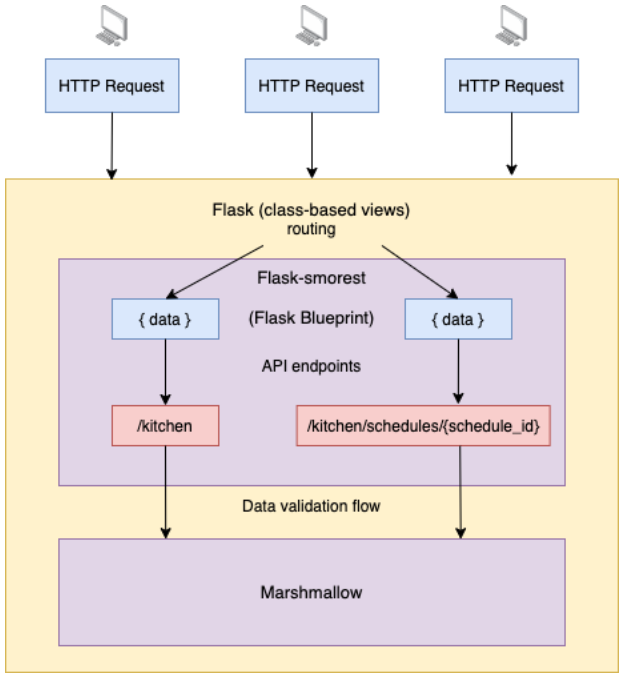


Figure 6.5 Architecture of an application built with Flask-smorest. Flask-smorest implements a typical Flask blueprint, which allows us to build and configure our API endpoints just as we would do in a standard Flask application.

Building APIs with Flask-smorest offers a similar experience to building them with FastAPI. In fact, FastAPI takes some inspiration from Flask in its design principles. However, there're some major differences between the two that we should be aware of:

- As we mentioned in section 2.3, FastAPI is built on top of Starlette, which implements an asynchronous web server framework. This means that with FastAPI we can build asynchronous applications, which in some cases can deliver better performance than applications built with more traditional frameworks such as Flask. To understand how

[4]

asynchronous servers can deliver better performance, head over to the sidebar "Asynchronous servers and performance".

- FastAPI uses Pydantic for data validation while Flask-smorest uses marshmallow. This means that with FastAPI we use native Python type hints to create validation rules for our data, while in marshmallow we use field classes. This means that with Pydantic we can leverage the growing ecosystem around type hints in Python to write cleaner and more reliable code.
- Flask allows us to implement API endpoints with class-based views. We'll see the details of how this works in section 6.8, but the gist of it is we can use a class to represent a URL path and its endpoints as methods of the class. Class-based views help you write more structured code and encapsulate the specific behavior of each URL path within its class. For example, if the endpoints under the /orders URL path needed to support special response headers, we could encapsulate this functionality within the Orders view class. In contrast, FastAPI only allows you to define endpoints using functions. Notice that Starlette does allow you to implement class-based routes, so this limitation of FastAPI may go away sometime in the future.

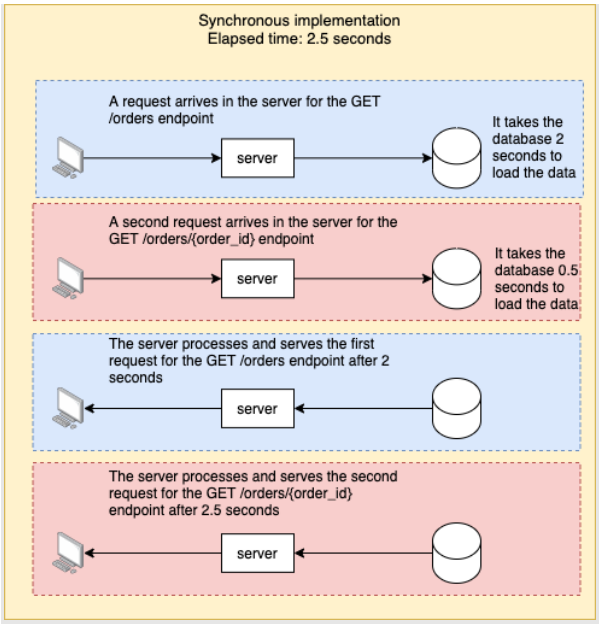
Asynchronous servers and performance

Do asynchronous servers deliver better performance? In many cases, yes. Notice that "more performant" here doesn't necessarily mean "faster". The point of asynchronous programming is to make sure that your server doesn't stay idle while waiting for a task to complete.

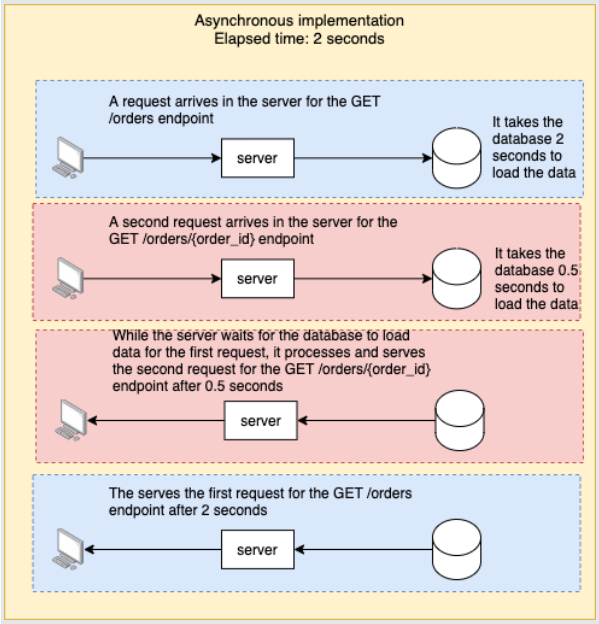
What does this mean for web applications? In your web server, some tasks will take more to complete than others. For example, in our orders API, the GET /orders endpoint may be slower than the GET /orders/{order_id} endpoint, since the first has to load a large list of records from the database, while the latter only returns one record. In this scenario, we'd want to make sure that, when our server is processing the GET /orders request, we can also serve GET/orders/{order_id} requests, and asynchronous programming helps us with that. In contrast, in a synchronous implementation, we'd have to wait for the GET /orders request to finish before we can begin processing the GET /orders/{order_id} request[5].

The figure in this sidebar illustrates this idea with a simplified example. A GET /orders request arrives first in the server and a GET /orders/{order_id} arrives right after. In this example, it takes the database 2 seconds to load the data for the GET /orders endpoint, while it takes only 0.5 seconds to load the data for the GET /orders/{order_id} endpoint. In the synchronous implementation, each task is blocking, which means we need to complete one before we can process the next, so the total elapsed time it takes to process both requests is 2.5 seconds.

[5] For an excellent explanation of how asynchronous programming works in Python, see Luciano Ramalho, *Fluent Python*, Sebastopol (O'Reilly), 2015, pp. 557-604. For an excellent discussion of the differences between concurrency, parallelism and asynchronous processing in Python see chapter 3 of Tiago Rodigues Antao, *High Performance Python for Data Analytics*, Shelter Island (Manning), expected 2021, in particular section 3.1 "Writing the scaffold of an asynchronous server" (https://livebook.manning.com/book/microservices-patterns/chapter-8/point-8620-53-297-0).



In a synchronous implementation, the server processes each request in order and it takes a total of 2.5 seconds to serve the two requests



In an asynchronous implementation, the server can process the second request while it waits for the database to load the data for the first request. Thanks to that, it takes a total of 2 seconds to process both requests

In contrast, in the asynchronous implementation the server can work on the GET /orders/{order_id} task while it waits for the database to load the data for the previous request. Thanks to that, the total elapsed time it takes to process both requests is 2 seconds.

These are the major differences between FastAPI and Flask-smorest, and other than that, it feels pretty much the same to build an API one or the other framework. With this covered, let's kick off the implementation of the kitchen API!

6.7 Initializing the web application for the API

In this section, we'll set up the environment to start working on the kitchen API, we'll create the entry point for the application and we'll add basic configuration for the web server. In doing so, you'll learn how to set up a project to work with flask-smorest and how to inject configuration objects into your Flask applications.

Flask-smorest is built on top of the Flask framework, so we'll lay out our web application following the patterns of a typical Flask application.

Create a folder called ch06/kitchen to contain the implementation of the kitchen APi. Go ahead and copy the `oas.yaml` file from the code repository into this folder as `ch06/kitchen/oas.yalml`. The `oas.yaml` file contains the API specification for the kitchen API that we'll implement in this chapter. `cd` into the `ch06/kitchen` folder, and run the following commands to install the dependencies that we'll need to proceed with the implementation:

```
$ pipenv install flask-smorest
```

> **If you want to ensure that you're installing the same version of the dependencies that I used when writing this chapter, copy the `Pipfile` and the `Pipfile.lock` files under the `ch06/kitchen` folder in the code repository into your local machine and run `pipenv install`.**

Also, run the following command to activate the environment:

```
$ pipenv shell
```

Now that we have the libraries that we need in place, let's create a file called `ch06/kitchen/app.py`. This file will contain an instance of the `Flask` application object, which will represent our web application. We'll also create an instance of the Flask-smorest's `Api` object, which will represent our API. Listing 6.4 shows the code needed in `app.py` to initialize the application.

Listing 6.4 Initialization of the Flask application object and the API object

```
from flask import Flask
from flask_smorest import Api

app = Flask(__name__)    #A

kitchen_api = Api(app)    #B
```

#A We get an instance of the Flask application object
#B We get an instance of the API object, which will represent the kitchen API

Flask-smorest requires some configuration parameters in order to work. For example, we should to specify the version of OpenAPI that we are using, the title of our API, and the version of our API. We need to load the configuration into the Flask application object. Flask offers different ways of doing this, but the most convenient method is loading configuration from a class. Let's create a file called `ch06/kitchen/config.py` to contain our configuration parameters. As shown in listing 6.5, within this file we'll create a `BaseConfig` class, which will contain generic configuration for the API which is unlikely to change depending on the environment. Further on, we can create a `Production` and a `Development` configuration classes, which will include configuration specific to those environments.

Listing 6.5 Configuration for the orders API

```
class BaseConfig:
    API_TITLE = 'Kitchen API'    #A
    API_VERSION = 'v1'    #B
    OPENAPI_VERSION = '3.0.3'    #C
    OPENAPI_JSON_PATH = 'oas.json'    #D
    OPENAPI_URL_PREFIX = '/'    #E
    OPENAPI_REDOC_PATH = '/redoc'    #F
    OPENAPI_REDOC_URL = 'https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js'
       #G
    OPENAPI_SWAGGER_UI_PATH = '/docs'    #H
    OPENAPI_SWAGGER_UI_URL = 'https://cdn.jsdelivr.net/npm/swagger-ui-dist/'    #I
```

#A The title of our API
#B The version of our API
#C The version of OpenAPI that we are using
#D Path to the dynamically generated specification in JSON
#E Prefix for the OpenAPI specification file
#F Path to the Redoc UI of our API (as we explained in the "What is a Swagger UI?" callout in section 6.3, Redoc is a UI style for REST interfaces)
#G Path to a script to be used to render the Redoc UI. In this case, we use a script served from a standard CDN
#H Path to the Swagger UI of our API
#I Path to a script to be used to render the Swagger UI. In this case, we use a script served from a standard CDN

Now that we have the configuration ready, we can pass it to the Flask application object. Listing 6.6 shows how we load configuration from a class into the application.

Listing 6.6 Loading configuration

```
from flask import Flask
from flask_smorest import Api

from config import BaseConfig    #A


app = Flask(__name__)
app.config.from_object(BaseConfig)    #B

kitchen_api = Api(app)
```

#A We import the BaseConfig class we defined earlier
#B We use the from_object method to load configuration from a class

With the entry point for our application ready and configured, let's move on to implementing the endpoints for the kitchen API!

6.8 Implementing the API endpoints

This section explains how we implement the endpoints of the kitchen API using Flask-smorest. Since Flask-smorest is built on top of Flask, building the endpoints for our API looks exactly the same as defining the endpoints of any other Flask application. In Flask, we register our endpoints using the `route` decorator of the Flask application object:

```
@app.route('/orders')
def process_order():
    pass
```

Using the `route` decorator works for simple cases, but for more complex application patterns, it's recommended to use Flask blueprints. Flask blueprints allow you to provide specific configuration for a group of URLs. To implement the kitchen API endpoints, we'll use the `Blueprint` class from Flask-smorest. Flask-smorest's `Blueprint` inherits from Flask's `Blueprint` class, so it provides all the functionality that comes with Flask blueprints, and it enhances it with additional functionality and configuration which makes it easier to generate API documentation, supply payload validation models, and add URL parameters validation, among other things.

We can use Blueprints' `route` decorators to decorate a function or a class that implements the functionality for a given endpoint or URL path. As you can see from figure 6.6, functions are convenient for URL paths that only accept one HTTP method. When a URL can accept multiple HTTP methods, it's more convenient to use class-based routes, which can be implemented by inheriting from the `MethodView` class from the `flask.views` module.

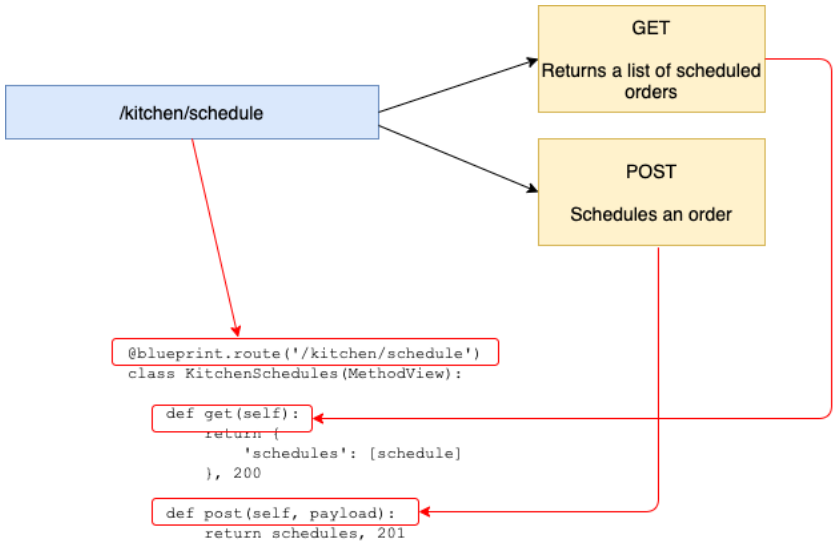


Figure 6.6 Class-based views. When a URL path exposes more than one HTTP method, it's more convenient to implement it as a class-based view, where the class methods implements each of the HTTP methods exposed. In this case, the KitchenSchedules class implements the /kitchen/schedule URL path, and its get and post methods implement their homonymous GET and POST HTTP methods.

As you can see in figure 6.7, using `MethodView`, we can represent a URL path as a class, and we can implement each of the endpoints of the URL path as methods named after the HTTP method they implement.

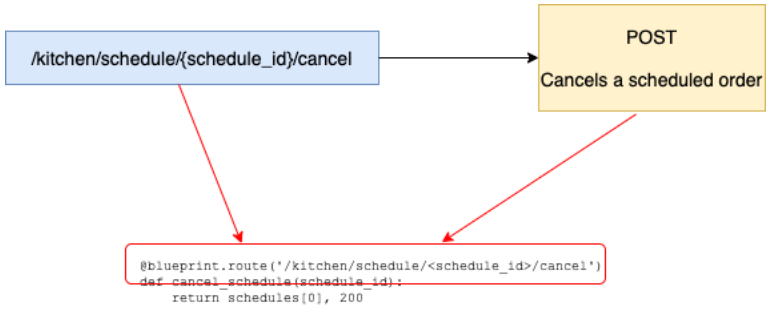


Figure 6.7 Function-based views. When a URL path exposes only one HTTP method, it's more convenient to implement it as a function-based view. In this case, the /kitchen/schedule/{schedule_id}/cancel only exposes a POST endpoint, so it's implement by the function cancel_schedule.

For example, if we have a URL path /kitchen that exposes a GET and a POST endpoints, we can implement the following class-based view:

```
class Kitchen(MethodView):

    def get(self):
        pass

    def post(self):
        pass
```

Listing 6.7 illustrates how we can implement the endpoints for the orders API using class-based views as well as function-based views. The first thing we do is getting an instance of Flask-smorest's `Blueprint`. The `Blueprint` will allow us to register our endpoints, add data validation to them, and to dynamically generate a Swagger UI from code, among other things. To instantiate `Blueprint`, we need to pass two required positional arguments: the name of the `Blueprint` itself, and the name of the module where the Blueprint's routes are defined. In this case, we are passing the name of the module using the module's `__name__` attribute, which resolves to the name of the file. We use `__name__` instead of a hardcoded string because the name of the file can change in the future, and we don't want to have to make changes to our blueprints based on that.

Once the Blueprint is instantiated, we register our URL paths with it using the route decorator. We use class-based routes for the `/kitchen/schedule` and the `/kitchen/schedule/{schedule_id}` paths since the expose more than one HTTP method, and we use function-based routes for the `/kitchen/schedule/{schedule_id}/cancel` and the `/kitchen/schedule/{schedule_id}/pay` paths because they only expose one HTTP method. At the moment we are returning a mock schedule object in each endpoint for illustration purposes, and we'll change that into a dynamic in-memory collection of schedules in section 6.12. The return value of each function is a tuple where the first element is the payload and the second element is the status code of the response.

Listing 6.7 Implementation of the endpoints of the orders API

```
import uuid
from datetime import datetime

from flask.views import MethodView
from flask_smorest import Blueprint


blueprint = Blueprint('kitchen', __name__, description='Kitchen API')    #A


schedules = [{    #B
    'id': str(uuid.uuid4()),
    'scheduled': datetime.now().timestamp(),
    'status': 'pending',
    'order': [
        {
            'product': 'capuccino',
            'quantity': 1,
            'size': 'big'
        }
```

```
    ]
}]


@blueprint.route('/kitchen/schedule')    #C
class KitchenSchedules(MethodView):    #D

    def get(self):    #E
        return {
            'schedules': [schedule]
        }, 200    #F

    def post(self, payload):
        return schedules, 201


@blueprint.route('/kitchen/schedule/<schedule_id>')    #G
class KitchenSchedule(MethodView):

    def get(self, schedule_id):    #H
        return schedules[0], 200

    def put(self, payload, schedule_id):
        return schedules[0], 200

    def delete(self, schedule_id):
        return '', 204


@blueprint.route('/kitchen/schedule/<schedule_id>/cancel')    #I
def cancel_schedule(schedule_id):
    return schedules[0], 200
```

#A We get an instance of the blueprint. The first argument is the name of the blueprint and the second argument is the name of the file here the bluepring is defined. We also pass in a description of the API paths
#B We define a hardcoded list of schedules that we can use while we're implementing our API functionality
#C To register a class or a function as a URL path, we simply use the Blueprint.route decorator
#D Because the /kitchen/schedule URL path exposes multiple HTTP methods, we implement is as a class-based view
#E Every method view in a class-based view is named after the HTTP method it implements
#F In Flask, we return both the payload and the status code from every function or method view
#G URL parameters are defined within angle brackets
#H If a function or method view takes a URL parameter, we must define it in the function signature
#I Since the /kitchen/schedule/<schedule_id>/cancel URL path only exposes one HTTP method, we implement it as a function-based view

Now that we have created the blueprint, we can register it with our API object in the app.py file. Listing 6.8 shows the changes necessary in app.py to register the blueprint.

Listing 6.8 Register blueprint with the APi object

```
from flask import Flask
from flask_smorest import Api

from api.api import blueprint    #A
from config import BaseConfig


app = Flask(__name__)
```

```
app.config.from_object(BaseConfig)

kitchen_api = Api(app)

kitchen_api.register_blueprint(blueprint)    #B
```

#A We import the blueprint we defined earlier
#B We register the blueprint with the kitchen API object

cd into the `ch06/kitchen` directory and run the application with the following command:

```
$ flask run --reload
```

Just like in `uvicorn`, the `--reload` flag runs the server with a watcher over your files, so that any changes you make to them trigger a server reload, instead of you having to stop and run the server.

If you go to the http://127.0.0.1/docs URL, you'll see an interactive Swagger UI dynamically generated from the endpoints we implemented earlier. You can also see the OpenAPI specification dynamically generated by Flask-smorest under http://127.0.0.1/openapi.json. At this stage in our implementation it's not possible to interact with the endpoints through the Swagger UI. Since we don't have Marshmallow models yet, Flask-smorest doesn't know how to serialize data and therefore doesn't return marshalled payloads. However, it's still possible to call the API using cURL and inspect the responses. If you run `curl http://127.0.0.1/kitchen/schedule`, you'll get a list with the mock schedule object we defined in the `api.py` module.

Things are looking good and it's now time to spice up the implementation by adding Marshmallow models. Move on to the next section to learn how to do that!

6.9 Implementing payload validation models

To leverage the data validation capabilities of Flask-smorest, we have to provide it with Marshmallow models. In this section, we learn to work marshmallow models by implementing the schemas of the kitchen APIO. The marshmallow models will help Flask-smorest validate our payloads and serialize our data. We already introduced marshmallow in section 5.3 when we used apispec to generate the OpenAPI specification for orders API from code. In this section, we'll apply what we learned in chapter 5 about marshmallow, and we'll learn additional features that will help us make our validations more robust.

As we saw earlier, the kitchen API contains three schemas: the `ScheduleOrderSchema` schema, which contains the details needed to schedule an order; the `GetScheduledOrderSchema`, which represents the details of a scheduled order; and `OrderItemSchema`, which represents a collection of items ordered by a customer. Listing 6.9 shows how to implement these schemas using marshmallow.

In section 5.3, we covered most of the classes and methods used in this implementation in chapter, so please refer to that section for further details. What's new in this implementation is the use of the `Meta` class and the presence of the `unknown` attribute, which we set to `EXCLUDE`. We can use the `Meta` class to customize and enhance our models. For example, we can specify what properties should be included in the serialization, what module should be used for serialization, or how to handle additional properties. Additional properties are handled with the `unknown` attribute.

Following our decision in section 6.3 to invalidate payloads which contain illegal properties, we set unknown to `EXCLUDE`. This will make marshmallow raise a validation error when a payload contains properties that haven't been defined in the schema.

Listing 6.9 Schema definitions for the Orders API

```
from marshmallow import Schema, fields, validate, EXCLUDE


class OrderItemSchema(Schema):
    class Meta:    #A
        unknown = EXCLUDE

    product = fields.String(required=True)
    size = fields.String(
        required=True, validate=validate.OneOf(['small', 'medium', 'big'])
    )
    quantity = fields.Integer(
        validate=validate.Range(1, min_inclusive=True), required=True
    )


class ScheduleOrderSchema(Schema):
    class Meta:
        unknown = EXCLUDE

    order = fields.List(fields.Nested(OrderItemSchema), required=True)


class GetScheduledOrderSchema(ScheduleOrderSchema):    #B
    id = fields.UUID(required=True)
    scheduled = fields.Integer(
        required=True, description='Date in the form of UNIX timestmap'
    )
    status = fields.String(
        required=True,
        validate=validate.OneOf(['pending', 'progress', 'cancelled', 'finished'])
    )
```

#A To customize the behavior of our marshmallow models, we define a nested Meta class. In this case, we use the Meta class to ban from the payload any properties that haven't been defined in the schema
#B We can use class inheritance to reuse the properties and definitions of an existing schema

Now that our validation models are ready, we can hook them up with our views. Listing 6.10 shows how we use the models to add validation for request and response payloads on our endpoints. Bear in mind we're only showing the relevant parts of the file that contain the changes. Omitted sections are marked with an ellipsis. To add a request payload validation to a view, we use the `Blueprint.arguments` decorator in combination with a marshmallow model. For response payloads, we use the `Blueprint.response` decorator in combination with a marshmallow model. If the response contains a list of objects, as in the GET /kitchen/schedule endpoint, we set the many argument in the model to True.

Notice that by decorating are methods and functions with the `Blueprint.response` decorator, we don't need to return a tuple of payload + status code anymore. Flask-smorest takes care of adding the status code for us. By default, Flask-smorest adds a 200 status `code` to our responses.

If we want to customize that, we simply need to specify the desired status code through the code parameter of the response decorator.

Notice that, while the `Blueprint.arguments` decorator validates and deserializes an request payload, the `Blueprint.response` decorator doesn't perform validation and only serializes the payload. We'll discuss this feature in more detail in 6.11, and we'll see how we can ensure that data is validated before being serialized.

Listing 6.10 Adding validation to the API endpoints

```
import uuid
from datetime import datetime

from flask.views import MethodView
from flask_smorest import Blueprint

from api.schemas import GetScheduledOrderSchema, ScheduleOrderSchema    #A

blueprint = Blueprint('kitchen', __name__, description='Kitchen API')

...

@blueprint.route('/kitchen/schedule')
class KitchenSchedules(MethodView):

    @blueprint.response(GetScheduledOrderSchema(many=True))    #B
    def get(self):
        return schedules    #C

    @blueprint.arguments(ScheduleOrderSchema)    #D
    @blueprint.response(GetScheduledOrderSchema, code=201)    #E
    def post(self, payload):
        return schedules[0]


@blueprint.route('/kitchen/schedule/<schedule_id>')
class KitchenSchedule(MethodView):

    @blueprint.response(GetScheduledOrderSchema)
    def get(self, schedule_id):
        return schedules[0]

    @blueprint.arguments(ScheduleOrderSchema)
    @blueprint.response(GetScheduledOrderSchema)
    def put(self, payload, schedule_id):
        return schedules[0]

    @blueprint.response(code=204)    #F
    def delete(self, schedule_id):
        return ''


@blueprint.response(GetScheduledOrderSchema)
@blueprint.route('/kitchen/schedule/<schedule_id>/cancel')
def cancel_schedule(schedule_id):
    return schedules[0]
```

#A We import the marshmallow schemas we defined earlier

#B We use the Blueprint.response decorator to register a validation schema for the response payloads of a function or method view
#C By decorating a method or function view with the Blueprint.response decorator, Flask-smorest takes care of adding the status code in the responses, and therefore we don't need to return anymore together with the payload
#D We use the Blueprint.arguments decorator to register a validation schema for the request payloads of a function or method view
#E By default, Flask-smorest adds a 200 status code to our responses. We can change that by setting the code parameter of the Blueprint.response decorator to the desired status code
#F To mark that an endpoint returns an empty response, we decorate it with an empty Blueprint.response. In this case, we are setting the status code of the response to 204, since this is a DELETE method

To see the effects of the new changes in the implementation, go back to the Swagger UI under http://127.0.0.1:5000/docs (also reproduced in figure 6.8 for your convenience). If you're running the server with the `--reload` flag, the changes will be automatically reloaded. Otherwise, stop the server and run it again. As you can see, Flask-smorest now recognizes the validation schemas that need to be used in the API, and therefore they're represented in the Swagger UI. If you now play around with the UI, for example by hitting the GET /kitchen/schedule endpoint, you'll be able to see the response payloads.

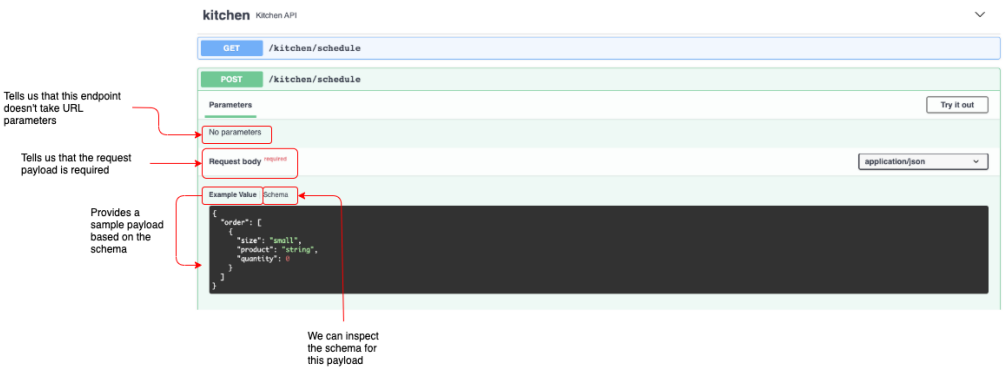


Figure 6.8 Schema documentation in Swagger UI. As we can see in this view for the POST /kitchen/schedule endpoint, the Swagger UI shows us the schema for the request payload of this endpoint, provides an example of it, and adds useful information, such as the lack of parameters in this endpoint.

The API is looking good and we are nearly finished with the implementation. The next step is adding URL query parameters to the GET /kitchen/schedule endpoint. Move on to the next section to learn how to do that!

6.10 Validating URL query parameters

In this section, we learn how to add URL query parameters to the GET /kitchen/schedule endpoint as per the specification. Remember that in section 6.2 we implemented URL query parameters for the orders API using FastAPI, and in this section you'll see that, regardless of the implementation details between FastAPI and Flask-smorest, the approach is similar and the resuts are the same.

URL query parameters are often used to filter the results of a GET endpoint. As shown in listing 6.11, the GET /kitchen/schedule endpoint accepts 3 URL query parameters:

- `progress` (Boolean): indicates whether an order is in progress.
- `limit` (integer): limits the number of results returned by the endpoint.
- `since` (date-time): filters results by the time when the orders were scheduled. A date in date-time format has the following structure: `YYYY-MM-DDTHH:mm:ssZ`. An example of this date format is: 2021-08-31T01:01:01Z. For more information on this format, see https://tools.ietf.org/html/rfc3339#section-5.6.

Listing 6.11 Specification for the GET /kitchen/schedule URL query parameters

```
paths:
  /kitchen/schedule:
    get:
      summary: Returns a list of orders scheduled for production
      parameters:
        - name: progress
          in: query
          description: Whether the order is in progress or not. In progress means it's in
       production in the kitchen.
          required: false
          schema:
            type: boolean
        - name: limit
          in: query
          required: false
          schema:
            type: integer
        - name: since
          in: query
          required: false
          schema:
            type: string
            format: 'date-time'
```

How do we implement URL query parameters in Flask-smorest? To begin with, we need to implement a new marshmallow model to represent them. Listing 6.12 shows how we implement the URL query parameters for the kitchen API using marshmallow. You can add the model for the URL query parameters to `ch06/api/schemas.py`, together with the other marshmallow models.

Listing 6.12 URL query parameters in marshmallow

```
from marshmallow import Schema, fields, validate, EXCLUDE

...

class GetKitchenScheduleParameters(Schema):
    progress = fields.Boolean()    #A
    limit = fields.Integer()
    since = fields.DateTime()
```

#A We define the fields of the URL query parameters in the exact same way that we define the fields of a payload

We register the schema for URL query parameters in the exact same way we register a schema for validating request payloads: using the Blueprint.arguments decorator. In this case, we need to specify that the properties defined in the schema are expected in the URL, so we set the location parameter to 'query', as shown in listing 6.13.

Listing 6.13 Add URL query parameters to GET /kitchen/schedule

```
import uuid
from datetime import datetime

from flask.views import MethodView
from flask_smorest import Blueprint

from api.schemas import (
    GetScheduledOrderSchema, ScheduleOrderSchema, GetKitchenScheduleParameters    #A
)

blueprint = Blueprint('kitchen', __name__, description='Kitchen API')


...


@blueprint.route('/kitchen/schedule')
class KitchenSchedules(MethodView):

    @blueprint.arguments(GetKitchenScheduleParameters, location='query')    #B
    @blueprint.response(GetScheduledOrderSchema(many=True))
    def get(self, parameters):    #C
        return schedules, 200

...
```

#A We import the model for the URL query parameters we defined earlier
#B We add the URL query parameters using the `Blueprint.arguments` decorator and setting the `location` parameter to `query`.
#C URL query parameter will be passed to our view function in the form of a dictionary, and we capture them with a single argument, which in this case we name parameters.

If you reload the Swagger UI, you'll see that the GET /kitchen/schedule endpoint now accepts three optional URL query parameters (shown in figure 6.9 for your convenience). These parameters should be passed on to our business layer, which will determine how to use them to return the list of results. We won't be implementing the service layer until chapter 7, so in the meantime listing 6.14 shows how we can use them to filter our in-memory list of schedules.

Listing 6.14 Use filters in GET /kitchen/schedules

```
...

@blueprint.route('/kitchen/schedule')
class KitchenSchedules(MethodView):

    @blueprint.arguments(GetKitchenScheduleParameters, location='query')
    @blueprint.response(GetScheduledOrderSchema(many=True))
    def get(self, parameters):
        if not parameters:    #A
```

```
            return schedules, 200

        query_set = [schedule for schedule in schedules]    #B

        in_progress = parameters.get(progress)    #C
        if in_progress is not None:
            if in_progress:
                query_set = [
                    schedule for schedule in schedules
                    if schedule['status'] == 'progress'
                ]
            else:
                query_set = [
                    schedule for schedule in schedules
                    if schedule['status'] != 'progress'
                ]

        since = parameters.get('since')
        if since is not None:
            query_set = [
                schedule for schedule in schedules
                if schedule['scheduled'] >= datetime.timestamp(since)
            ]

        limit = parameters.get('limit')
        if limit is not None and len(query_set > limit):    #F
            query_set = query_set[:limit]

        return query_set, 200    #G

...
```

#A URL query parameters come in the form of a dictionary. If the user didn't set any URL query parameters, the dictionary will be empty and therefore and it'll evaluate to False

#B If the user set any URL query parameters, we'll filter the list of schedules. To do that, we make a copy of the list into a variable named query_set.

#C We check for the presence of each URL query parameter by accessing its value using the dictionary's get method. If the value isn't there, this will return None, and we can use that in the next line to check whether the parameter was set

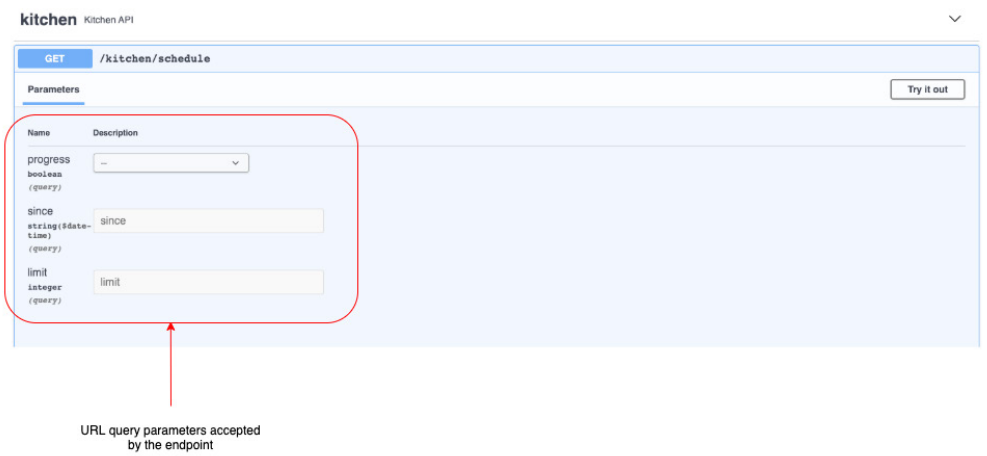


Figure 6.9 URL query parameters in Swagger UI. As we can see in this view for the GET /kitchen/schedule endpoint, the Swagger UI shows us the URL query parameters accepted by the endpoint, and it offers form fields that we can fill in to experiment with different values.

Now that we know how handle URL query parameters with Flask-smorest, there's one more topic we need to cover, and that is data validation before serialization. Move on to the next section to learn more about this!

6.11 Validating data before serializing the response

Now that we have schemas to validate our request payloads and we have hooked them up with our routes, we have to ensure that our response payloads are also validated. In this section, we learn how to use Marshmallow models to validate data. We'll use this functionality to validate our response payloads, but you could use the same approach to validate any kind of data, such as configuration objects.

When a payload is sent to the user in the body of a response, Flask-smorest serializes the payload using marshmallow. However, as shown in figure 6.10, before serializing the payload, marshmallow doesn't validate that the payload is correctly formed[6].

[6] Before version 3.0.0 marshmallow used to perform validation before serialization (see the changelog: https://github.com/marshmallow-code/marshmallow/blob/dev/CHANGELOG.rst#300-2019-08-18).

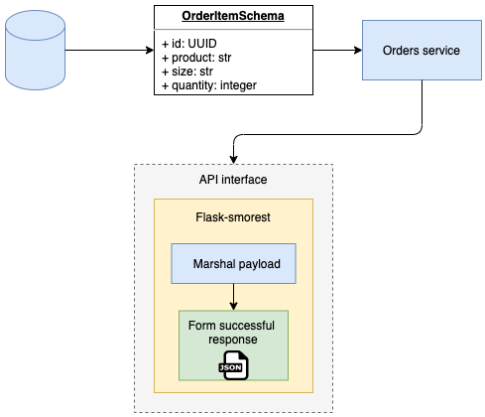


Figure 6.10 Workflow of a data payload with the Flask-smorest framework. Response payloads are supposed to come from a "trusted-zone" and therefore Marshmallow doesn't validate them before marshalling.

This is another difference with FastAPI, which does validate our data before it's serialized for a response, as you can see in figure 6.11.

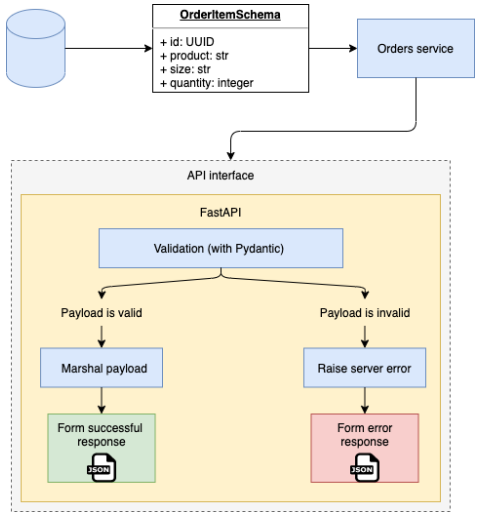


Figure 6.11 Workflow of a data payload with the FastAPI framework. Before marshalling and forming a response, FastAPI validates that the payload conforms to the specified schema.

The fact that marshmallow doesn't perform validation before serialization is not necessarily undesirable. In fact, it can be argued that it's a desirable behavior, as it decouples the task of

serializing from the task of validating the payload. There are two rationales for which marshmallow doesn't perform validation before serialization[7]:

1. it improves performance, since validation is slow;
2. data coming from the server is supposed to be trusted data and therefore shouldn't require validation.

The reasons adduced by the maintainers of marshmallow to justify this design decision are fair. However, if you've worked long enough with APIs, and websites in general, you know there's generally very little to be trusted even from within your own system.

> **ZERO-TRUST APPROACH FOR ROBUST APIS.** API integrations fail due to the server sending the wrong payload as much as they fail due to the client sending malformed payloads to the server. Whenever possible, it's good practice to take a zero-trust approach to our systems design.

The data that we send from the kitchen API comes from a database. In chapter 7, we'll learn patterns and techniques to ensure that our database contains the right data in the right format. However, the database is an independent component of our platform, and even under the strictest access security measures, there's always a chance that the wrong data ends up in the database. As unlikely as this is, we don't want to blow up our user experience in the event that it happens, and validating our data before serializing it will help us with that.

Thankfully, marshmallow makes it easy to perform validation. To validate our data, we simply need to get an instance of the schema we intend to validate against, and use the validate method, passing in the data we want to validate. Notice that validate won't raise an exception if errors are found. Instead, it'll return a dictionary containing any errors found. If no errors are found, it'll return an empty dictionary. To get a feeling for how this works, open a Python shell by typing `python` in the terminal and run the following code:

```
>>> from api.schemas import GetScheduledOrderSchema
>>> GetScheduledOrderSchema().validate({'id': 'asdf'})
{'order': ['Missing data for required field.'], 'scheduled': ['Missing data for required
       field.'], 'status': ['Missing data for required field.'], 'id': ['Not a valid UUID.']}
```

We pass a malformed representation of a schedule containing only the `id` field, and marshmallow helpfully reports that the `order`, `scheduled`, and the `status` fields are missing, and that the `id` field is not a valid UUID. We can use this information to raise a helpful error message in the server, as shown in listing 6.15.

Listing 6.15 Validate data before serialization

```
import uuid
from datetime import datetime

from flask.views import MethodView
from flask_smorest import Blueprint
from marshmallow import ValidationError    #A

...
```

[7] See the full discussion here: https://github.com/marshmallow-code/marshmallow/issues/682.

```
@blueprint.route('/kitchen/schedule')
class KitchenSchedules(MethodView):

    @blueprint.arguments(GetKitchenScheduleParameters, location='query')
    @blueprint.response(GetScheduledOrderSchema(many=True))
    def get(self, args):
        errors = GetScheduledOrderSchema().validate(schedule)    #B
        if errors:
            raise ValidationError(errors)    #C
        return schedules, 200

...
```

#A We import the ValidationError class from marshmallow, which will automatically format the error message into an appropriate HTTP response when raised
#B Schema.validate always returns a dictionary. If validation errors were found, the dictionary will contain those errors, otherwise the dictionary will be empty and therefore it'll evaluate to False in the next line
#C Schema.validate found errors, we raise a ValidationError exception

Please be aware that there are known issues with validation in marshmallow, especially when your models contain complex configuration for determining which fields should be serialized and which fields shouldn't (see https://github.com/marshmallow-code/marshmallow/issues/682 for additional information). Also, take into account that validation is known to be a slow process, so if you are handling large payloads you may to use a different tool to validate your data, or perhaps only validate a subset of your data, or skip validation altogether. However, whenever possible, you're better off performing validation on your data.

This concludes the implementation of the functionality of the kitchen API. However, the API is still returning the same mock schedule across all endpoints. Before concluding this chapter, let's add a minimal implementation of an in-memory list of schedules so that we can make our API dynamic. This will allow us to verify that all endpoints are functioning as intended.

6.12 Implementing an in-memory list of schedules

In this section we implement a simple in-memory representation of schedules so that we can obtain dynamic results from the API. By the end of this section, we'll be able to schedule orders, update them, and cancel them through the API. As the schedules are managed in an in-memory list, any time the server is reloaded or stopped we'll lose information from our previous session. In the next chapter we'll address this problem by adding a persistence layer to our service.

Our in-memory collection of schedules will be represented by a Python list, and we'll just add and remove from it from our function views and methods. Listing 6.16 shows the changes that we need to make to `ch06/kitchen/api/api.py` to make this possible.

Listing 6.16 In-memory implementation of schedules

```
...

schedules = []    #A


def validate_schedule(schedule):    #B
    errors = GetScheduledOrderSchema().validate(schedule)
```

```
    if errors:
        raise ValidationError(errors)


@blueprint.route('/kitchen/schedule')
class KitchenSchedules(MethodView):

    @blueprint.arguments(GetKitchenScheduleParameters, location='query')
    @blueprint.response(GetScheduledOrderSchema(many=True))
    def get(self, args):
        ...

    @blueprint.arguments(ScheduleOrderSchema)
    @blueprint.response(GetScheduledOrderSchema, code=201)
    def post(self, payload):
        payload['id'] = str(uuid.uuid4())    #C
        payload['scheduled'] = datetime.now().timestamp()
        payload['status'] = 'pending'
        schedules.append(payload)
        validate_schedule(schedule)
        return payload


@blueprint.route('/kitchen/schedule/<schedule_id>')
class KitchenSchedule(MethodView):

    @blueprint.response(GetScheduledOrderSchema)
    def get(self, schedule_id):
        for schedule in schedules:
            if schedule['id'] == schedule_id:
                validate_schedule(schedule)
                return schedule
        abort(404, description=f'Resource with ID {schedule_id} not found')    #D

    @blueprint.arguments(ScheduleOrderSchema)
    @blueprint.response(GetScheduledOrderSchema)
    def put(self, payload, schedule_id):
        for schedule in schedules:
            if schedule['id'] == schedule_id:
                schedule.update(payload)    #E
                validate_schedule(schedule)
                return schedule
        abort(404, description=f'Resource with ID {schedule_id} not found')

    @blueprint.response(code=204)
    def delete(self, schedule_id):
        for index, schedule in enumerate(schedules):
            if schedule['id'] == schedule_id:
                schedules.pop(index)    #F
                return
        abort(404, description=f'Resource with ID {schedule_id} not found')


@blueprint.response(GetScheduledOrderSchema)
@blueprint.route('/kitchen/schedule/<schedule_id>/cancel')
def cancel_schedule(schedule_id):
    for schedule in schedules:
        if schedule['id'] == schedule_id:
            schedule['status'] = 'cancelled'    #G
```

```
            validate_schedule(schedule)
            return schedule
    abort(404, description=f'Resource with ID {schedule_id} not found')
```

#A Since we are going to be able to add and remove schedules dynamically, we initializes schedules with an empty list
#B We refactor the functionality we implemented earlier to validate a schedule into a function so that we can reuse it in other method or function views
#C When a schedule payload arrives through the POST method, we need to enhance its properties by setting an ID, scheduled time, and its status
#D If a scheduled isn't found with the provided schedule_id, we return a 404 Not Found response. In Flask, we do this by using the abort function
#E When a user updates a schedule with new details, we use the payload to update the schedule's properties
#F We remove the schedule from the list and return an empty response
#G We set the status of the schedule to cancelled

If you reload the Swagger UI and test the endpoints, you'll see you're now able to add schedules, update them, cancel them, list and filter them, get their details, and delete them. This concludes our journey to implement REST APIs using Python. In the next chapter, we'll learn patterns to implement the rest of the service following best practices and useful patterns. Things are spicing up!

6.13 Summary

- FastAPI and Flask-smorest are two highly popular API development frameworks for Python.
    - FastAPI is built on top of Starlette, an asynchronous server framework, which means in some cases it can deliver better performance than traditional frameworks like Flask
    - Flask-smorest is built on top of Flask and it works as a Flask blueprint, which means you can use the software patterns that you'd use in a standard Flask application and leverage the whole Flask ecosystem.
- FastAPI uses Pydantic for data validation, while Flask-smorest uses marshmallow.
    - Pydantic is a very popular data validation library for Python which uses type hints to create validation rules. Type hints were introduced in Python 3.5 and are widely used today, which makes Pydantic models look and feel like modern and elegant Python code
    - Marshmallow is a battle-tested data validation framework which has been around for a long time[8]. When you use Marshmallow, you leverage years of improvements and optimizations to this library.
- FastAPI validates response payloads before marshalling while Flask-smorest takes the view that everything that comes from the server should be trusted and therefore doesn't require validation. However, you can still use Marshmallow's validation methods to validate your payloads.
- When validating payloads, you have to balance the benefits of the tolerant reader pattern with the risks of integration failures due to bugs like typos. For large and complex payloads,

[8] The first release was made on the 11th of November of 2013 (https://github.com/marshmallow-code/marshmallow/tags?after=0.2.0).

it may not be worthwhile or feasible to validate all the data

- Both FastAPI and Flask-smorest can generate API documentation from code and can render a Swagger UI or a Redoc UI, which we can use to test and visualize our implementation
- When working with FastAPI, there's no need to implement all the validation schemas by hand. Instead, you can use datamodel-code-generator to automatically generate them from the OpenAPI specification. This will save you time when working with large specifications

7

Service implementation patterns for microservices

This chapter covers

- How hexagonal architecture helps us design loosely coupled services
- Implementing the business layer for a microservice
- Implementing database models using SQLAlchemy
- How the Repository pattern helps us decouple the data layer from the business layer
- How the Unit of Work pattern helps us ensure the atomicity of all transactions
- How the dependency inversion principle helps us build software which is resilient to changes
- How the inversion of control principle and the dependency injection pattern help us decouple components which are dependent on each other

In this chapter, we'll learn how to implement the business layer of a microservice. In previous chapters, we learned how to design and implement REST APIs. In those implementations, we used an in-memory representation of the resources managed by the service. We took that approach to keep the implementation simple and allow ourselves to focus on the API layer of the service.

In this chapter, we'll complete our implementation of the orders service by adding a business layer and a data layer. The business layer will implement the capabilities of the orders service, such as taking orders, processing their payments, or scheduling them for production. For some of these tasks, the orders service requires the collaboration of other services, and we'll learn useful patterns to handle those integrations.

The data layer will implement the data management capabilities of the service. The orders service owns and manages data about orders, so we'll implement a persistent storage solution and an interface to it. However, as a gateway to the users regarding the lifecycle of an order, the orders service also needs to fetch data from other services; for example, to keep track of the order

during production and delivery. We'll also learn useful patterns to handle access from those services.

To articulate the implementation patterns of the service, we'll also cover elements of the architectural layout required to keep all pieces of our microservices loosely coupled. Loose coupling will help us ensure that we can change the implementation of a specific component without having to make changes to other components that rely on it. It'll also make our codebase generally more readable, maintainable, and testable.

The code for this chapter is available under chapter ch07 in the repository provided with this book.

7.1 Hexagonal architectures for microservices

This chapter introduces the concept of hexagonal architecture and how we'll apply it to the design of the orders service. In section 2.2, we introduced a variant of the Model-View-Controller (MVC) pattern to help us organize the components of our application in a loosely coupled way. In this section, we'll take this idea further by applying the concept of hexagonal architecture to our design.

In 2005, Alistair Cockburn introduced the concept of hexagonal architecture, also called architecture of ports and adapters, as a way to help software developers structure their code into loosely coupled components[1]. As you can see in figure 7.1, the idea behind the hexagonal or ports and adapters architecture is that, in any application, there's a core piece of logic that implements the capabilities of a service, and around that core we "attach" adapters that help the core communicate with external components. For example, a web API is an adapter that helps the core communicate with web clients over the Internet. The same goes for a database. A database is simply an external component that helps a service persist data. We should be able to swap the database if we wanted to, and the service would still be the same. Therefore, the database is also an adapter.

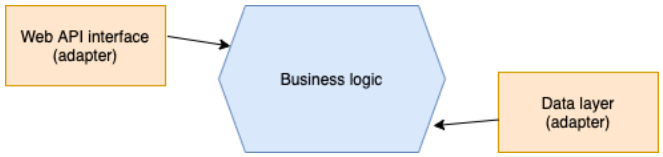


Figure 7.1 In hexagonal architecture, we distinguish a core layer in our application, the business layer, which implements the service's capabilities. Other components, such as a web API interface or a database, are considered adapters which depend on the business layer

How does this help us build loosely coupled services? The hexagonal architecture requires that we keep the core logic of the service and the logic for the adapters strictly separated. In other words, the logic that implements our web API layer shouldn't interfere with the implementation of the core business logic. And the same goes for the database: regardless of the technology we

[1] Alistair Cockburn, "Hexagonal architecture" (https://alistair.cockburn.us/hexagonal-architecture/ [accessed 11/11/2020]). You may be wondering why hexagonal and not pentagonal or heptagonal. As Alistair points out, it "is not a hexagon because the number six is important", but because it helps to highlight visually the idea of a core application communicating with external components through ports (the sides of the hexagon), and it allows us to represent the two main sides of an application: the public facing side (web components, APIs, etc.), and the internal side (databases, third party integrations, etc.).

choose, its design and idiosyncrasies, it shouldn't interfere with the core business logic. How do we achieve that? By building ports (read interfaces) between the core business layer and the adapters. Later in this chapter, we'll learn some design patterns that will help us design of those ports or interfaces.

When working out the relationship between the core business logic and the adapters, we apply the dependency inversion principle, which states that (see figure 7.2 for clarification)

- High-level modules shouldn't depend on low-level details. Instead, both should depend on abstractions, such as interfaces. For example, when saving data, we want to do it through an interface that doesn't require us to understand the specific implementation details of the database. Whether it's a SQL or a NoSQL database, or a cache store, the interface should be the same.
- Abstractions shouldn't depend on details. Instead, details should depend on abstractions[2]. For example, when designing the interface between the business layer and the data layer, we want to make sure that the interface doesn't change depending on the implementation details of the database. Instead, we make changes to the data layer to make it work with the interface. In other words, the data layer depends on the interface, and not the other way around.

DEFINITION: DEPENDENCY INVERSION PRINCIPLE. The Dependency Inversion Principle encourages us to design our software against interfaces, and make sure we don't create depedencies between the low-level details of our components.

The concept of dependency inversion often appears together with the concepts of inversion of control and dependency injection. These are related by different concepts. As we'll see in section 7.5, the inversion of control principle recommends us to supply code dependencies through the execution context (also called inversion of control container). To supply such dependencies, we can use the dependency injection pattern, which we'll describe in section 7.5.

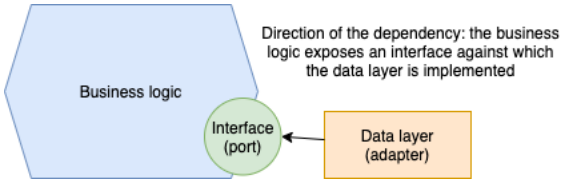


Figure 7.2 We apply the inversion of dependency principle to determine which components drive the changes. In hexagonal architecture, this means that our adapters will depend on the interface exposed by the core business layer.

What does this mean in practice? It means we should make the adapters depend on the interface exposed by the core business logic. That is, it's OK for our API layer to know about the core business logic's interface, but it's not OK for our business logic to know specific details of our API layer, not to say low-level details of the HTTP protocol. The same goes for the database: our data layer should know how the application works and how to accommodate the application's needs

[2] Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2003, pp. 127-131.

to our choice of storage technology, but the core business layer should know nothing specific about the database. Our business layer will expose an interface, and all other components will be implemented against it.

So, what exactly are we inverting with the inversion of dependency principle? This principle inverts the way we think about software. Instead of the more conventional approach of building the low-level details of our software first, and then building interfaces on top of them, the dependency inversion principle encourages us to think of the interfaces first, and then build the low-level details against those interfaces[3].

As you can see in figure 7.3, when it comes to the orders service, this means that we'll have a core package which implements the capabilities of the service. This includes the ability to process an order and its payment, to schedule its production or to keep track of its progress. The core service package will expose interfaces for other components of the application. Another package implements the web API layer, and our API modules will use functions and classes from the business layer interface in order to serve the requests of our users. And another package implements the data layer, which knows how to interact with the database and return business objects for the core business layer.

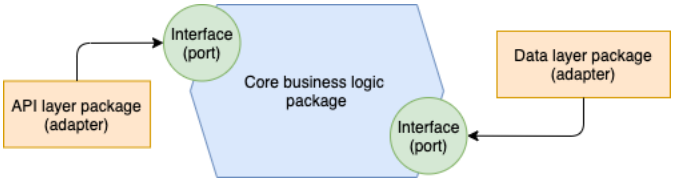


Figure 7.3 The orders service consist of three packages: a core business logic which implements the capabilities of the service, an API layer which allows clients to interact with the service over HTTP, and a data layer which allows the service to interact with the database. The core business logic exposes interfaces against which the API layer and the data layer are implemented.

Now that we know how we are going to structure the application, it's time to start implementing it! In the next section, we'll set up the environment to start working on the service.

7.2 Setting up the environment and the project structure

In this section, we set up the environment to work on the orders service and we lay out the high-level structure of the project. As in previous chapters, we'll use Pipenv to manage our dependencies. Run the following commands to set up a Pipenv environment and activate it:

```
$ pipenv --three
$ pipenv shell
```

We'll install our dependencies as we need in the following sections. Or if you prefer so, copy over the Pipfile from the repository under `ch07/Pipfile` and run `pipenv install`.

[3] For an excellent introduction to the dependency inversion principle, see Eric Freeman, Elizabeth Robson, Kathy Sierra, and Bert Bates, *Head First Design Patterns*, Sebastopol (CA), O'Reilly, 2014, pp. 141-143.

Our service implementation will live under a folder named orders, so go ahead and create it. To reinforce the separation of concerns between the core business layer and the API and database adapters, we'll implement each of them in different directories, as shown in figure 7.4. The business layer will live under `orders/orders_service`. Since the API layer is a web component, it will live under `orders/web`. `orders/web` contains web adapters for the orders service. In this case, we are only including one type of web adapter, namely a REST API, but nothing prevents you from adding a web adapter that returns dynamically rendered content from the server, as you would do in a more traditional Django application.

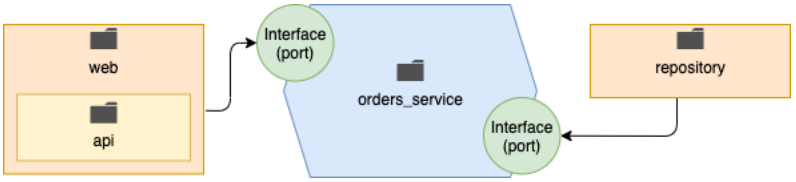


Figure 7.4 To reinforce the separation of concerns, we implement each layer of the application in different directories: orders_service for the core business layer; repository for the data layer; and web/api for the APi layer.

The data layer will live under `orders/repository`. Repository might look like an unlikely name for our data layer, but we're choosing this name because we'll implement the repository pattern to interface with our data. This concept will become clearer in section 7.4. In chapters 2 and 6 we covered the implementation of API layer, so go ahead and copy over the files from the GitHub repository under `ch07/order/web` into your local directory. Notice that the API implementation has been adapted for this chapter. Listing 7.1 shows the current structure of our project.

Listing 7.1 High-level structure of the orders service

```
.
├── Pipfile    #A
├── Pipfile.lock
└── orders    #B
    ├── orders_service    #C
    ├── repository    #D
    └── web    #E
        ├── api    #F
        │   ├── api.py
        │   └── schemas.py
        └── app.py    #G
```

#A Pipfile contains the dependencies that we need to run this project.
#B The full implementation of the orders service, including the business layer, the API, and the database layer, goes under the orders directory
#C This folder will contain the implementation of the business layer
#D This folder contains everything related to the data layer, including our database models and the orders repository
#E This folder contains web adapters that interface with the orders service, such as the REST API
#F This folder contains our REST API implementation for the orders service
#G This file contains the instance of our web application object. If we had multiple web adapters, all of them would share this object

Now that our project is set up and ready to go, it's time to get on with the implementation! Move on to the next section to learn how to add database models to the service!

7.3 Implementing the database models

In the previous section, we learned how we'll structure our project into three different layers: the core business layer, the API layer, and the data layer. This structure reinforces the separation of concerns among each layer, as recommended by the hexagonal architecture pattern that we learned in section 7.1. Now that we know how we'll structure our code, it's time to focus on the implementation! In this section, we'll define the database models for the orders service, that is we'll design the database tables and their fields. We start our implementation from the database since it will facilitate the rest of the exposition in this chapter. In a real-world context, you might start with the business layer, mocking the data layer, and iterate back and forth between each layer until you're done with the implementation. Just bear in mind that the linear approach we are taking in this chapter is not meant to reflect the actual development process, but is instead intended to illustrate concepts that we want to explain.

To keep things simple in this chapter, we'll use SQLite as our database engine. SQLite is a file-based relational database system. To use it, we don't need to set up and run a server, as we would with PostgreSQL or MySQL, and there's no configuration needed to start using it. Python's core library has built-in support for interfacing with SQLite, which makes SQLite a suitable choice for quick prototyping and experimentation before we are ready to move on to a production-ready database system.

We won't manage our connection to the database and our queries manually. That is, we won't be writing our own SQL statements to interact with the database. Instead, we'll use SQLAlchemy. SQLAlchemy is by far the most popular Object Relational Mapper (ORM) in the Python ecosystem. An ORM is a framework is an implementation of the Active Record Pattern which allows us to map the tables in our database to objects.

> **DEFINITION: ACTIVE RECORD PATTERN.** Active Record is an object wrapper around database tables and rows. It encapsulates database operations in the form of class methods, and it allows us to access data fields through class attributes.

As you can see in figure 7.5, using an ORM makes it easier to manage our data since it gives us a class interface to the tables in the database. This allows us to leverage the benefits of object-oriented programming, including the ability to add custom methods and properties to our database models that enhance their functionality and encapsulate their behavior.

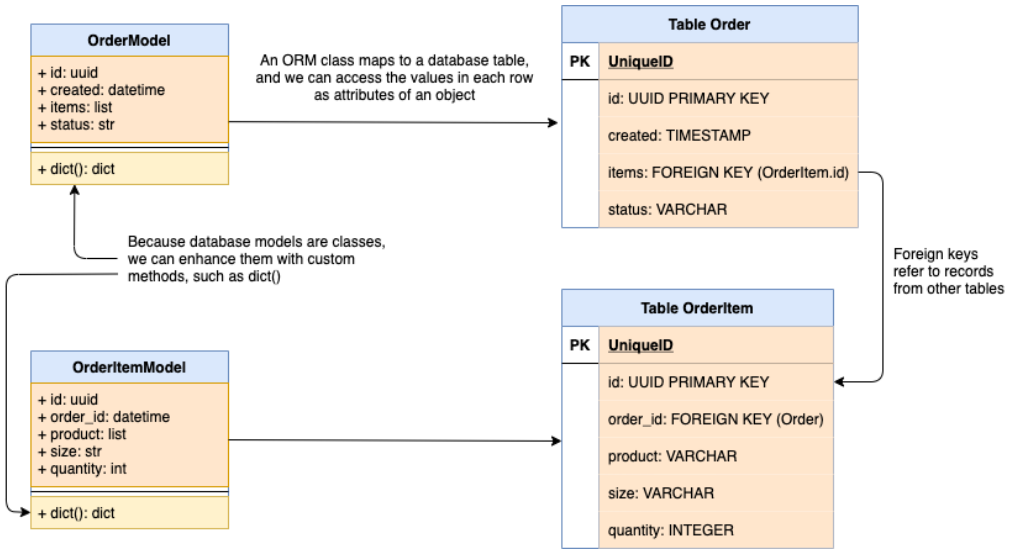


Figure 7.5 Using an ORM, we can implement our data models as classes that map to database tables. Since the models are classes, we can enhance them with custom methods to add new functionality

Over time, our database models will change, and we need to be able to keep track of those changes. Changing the schema of our database is called a migration. As our database evolves, we'll accumulate more and more migrations. We need to keep track of our migrations, since that allows us to reliably replicate the database schema in different environments, and to rollout database changes to production with confidence. To manage this complex task, we'll use Alembic. Alembic is a schema migration library that integrates seamlessly with SQLAlchemy.

Let's start by installing both libraries by running the following command:

```
$ pipenv install sqlalchemy alembic
```

Before we start working on our database models, let's set up Alembic. Run the following command to create a migrations folder, which will contain the history of all migrations in our database:

```
$ alembic init migrations
```

This creates a folder called `migrations`. The `migrations` folder comes with a configuration file called `env.py` and a `versions` directory. `versions` will contain the migration files. The setup command also creates a configuration file called `alembic.ini`. To make alembic work with an SQLite database, open `alembic.ini`, find a line which contains a declaration for the `sqlalchemy.url` variable, and replace it with the following content:

```
sqlalchemy.url = sqlite:///orders.db
```

COMMIT THE FILES GENERATED BY ALEMBIC The `migrations` folder contains all the information required to manage our database schema changes, so you should commit this folder, as well as `alembic.ini`. This will allow you to replicate the database set up in new environments.

In addition to this, open `migartions/env.py` and find the lines with the following content[4]:

```
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
target_metadata = None
```

And replace them with the following content:

```
from orders.repository.models import Base
target_metadata = Base.metadata
```

Next, we'll implement our database models! Before we jump straight into the implementation, let's pause for a moment to think about how many models we'll need, and the properties that we should expect each model. The core object of the orders service is the order. Users place, pay, update, or cancel orders. Orders have a lifecycle, and we'll keep track of it through a `status` property. We'll use the following list of properties to define our order model:

- User ID: the ID of the user who placed the order. Since we don't have authentication yet, this will be a random number for now. In chapter 12, we'll add authentication to our services, and we'll create a model for users.
- ID: unique ID for the order. We'll give it the format of a Universally Unique Identifier (UUID). Using UUIDs instead of incremental integers is quite common these days. UUIDs work well in distributed systems, and they help to hide information about the number of orders that exist in the database from our users.
- Creation date: when the order was placed.
- Items: the list of items included in the order and the amounts of each product. Since an order can have any number of items linked to it, we'll use a different model for items, and we'll create a one-to-many relationship between the order and the items.
- Status: the status of the order throughout the system. An order can have the following statuses:
    - Created: the order has been placed.
    - Paid: the order has been successfully paid.
    - Progress: the order is being produced in the kitchen.
    - Cancelled: the order has been cancelled.
    - Dispatched: the order is being delivered to the user.
    - Delivered: the order has been delivered to the user.
- Schedule ID: the ID of the order in the kitchen service. This ID is created by the kitchen service after scheduling the order for production, and we'll use it to keep track of its progress in the kitchen.
- Delivery ID: the ID of the order in the delivery service. This ID is created by the delivery service after scheduling it for dispatchment, and we'll use it to keep track of its progress

[4] The shape and format of this file may change over time, but for reference, at the time of this writing those lines are 18-20.

during delivery.

When users place an order, they add any number of items to the order. Each item contains information about the product selected by the user, the size of the product, and the amount of it that the user wishes to purchase. There's a one-to-many relationship between orders and items, and therefore we'll implement a model for items and link them with a foreign key relationship. The item model will have the following list of attributes:

- ID: a unique identifier for the item in UUID format.
- Order ID: a foreign key representing the ID of the order the item belongs to. This is what allows us to connect items and orders that belong together.
- Product: the product selected by the user.
- Size: the size of the product.
- Quantity: the amount of the product that the user wishes to purchase.

Our SQLAlchemy models will live under the `orders/repository` folder, which we created to encapsulate our data layer, in a file called `orders/repository/models.py`. We'll use these classes to interface with the database, and rely SQLAlchemy to translate these models into their corresponding database tables behind the scenes. Listing 7.2 shows the definition of the database models for the orders service. The first thing we need to do is create a declarative base model by using SQLALchemy's `declarative_base()` function. The declarative base model is a class that can map ORM classes to database tables and columns, and therefore all our database models must inherit from it. We map class attributes to specific database columns by setting them to instances of SQLAlchemy's `Column` class.

To map an attribute to another model, we use SQLAlchemy's `relationship()` function. In listing 7.2, we use `relationship()` to create a one-to-many relationship between `OrderModel`'s `item` attribute and the `OrderItemModel` model. This means that we can access the list of items in an order through `OrderModel`'s `item` attribute. Each item also maps to the order it belongs to through the `order_id` property, which is defined as a foreign key column. Furthermore, `relationship()`'s `backref` argument allows us to access the full order object from an item directly through a property called `order`.

Since we want our IDs to be in UUID format, we create a function that SQLAlchemy can use to generate the value. If we later switch to a database engine with built-in support for generating UUID values, we'll leave it to the database to generate the IDs. Each database model is enhanced with a `dict()` method which allows us to output the properties of a record in dictionary format. Since we'll use this method to translate database models to business objects, the `dict()` method only returns the properties that are relevant for the business layer.

Listing 7.2 SQLAlchemy models for the orders service

```
import uuid
from datetime import datetime

from sqlalchemy import Column, Integer, String, ForeignKey, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()    #A
```

```
def generate_uuid():    #B
    return str(uuid.uuid4())


class OrderModel(Base):    #C
    __tablename__ = 'order'    #D

    id = Column(String, primary_key=True, default=generate_uuid)    #E
    user_id: Column(String, nullable=False)
    items = relationship('OrderItemModel', backref='order')    #F
    status = Column(String, nullable=False, default='created')
    created = Column(DateTime, default=datetime.utcnow)
    schedule_id = Column(String)
    delivery_id = Column(String)

    def dict(self):    #G
        return {
            'id': self.id,
            'items': [item.dict() for item in self.items],    #H
            'status': self.status,
            'created': self.created,
            'schedule_id': self.schedule_id,
            'delivery_id': self.delivery_id,
        }


class OrderItemModel(Base):
    __tablename__ = 'order_item'

    id = Column(String, primary_key=True, default=generate_uuid)
    order_id = Column(Integer, ForeignKey('order.id'))
    product = Column(String, nullable=False)
    size = Column(String, nullable=False)
    quantity = Column(Integer, nullable=False)

    def dict(self):
        return {
            'id': self.id,
            'product': self.product,
            'size': self.size,
            'quantity': self.quantity
        }
```

#A declarative_base is a factory method that creates a base class for declarative class definitions, i.e. our model classes.
#B Custom function to create random UUIDs for our models
#C All our declarative class definitions must inherit from Base so that SQLAlchemy can map them to database schemas
#D Name of the table that maps to this class
#E Every class property maps to a database column by using the Column class
#F We use relationship() to create a one-to-many relationship the OrderItemModel model
#G Custom method to output the properties needed from the model by the business layer
#H Each item is an object, so we need to call the dict() method on each object to get their dictionary representation as well

To apply the models to the database, run the following command:

```
$ PYTHONPATH=`pwd` alembic revision --autogenerate -m "Initial migration"
```

This will create a migration file under `migrations/versions`. You should commit your migration files and keep them in your git repository, since they'll allow you to recreate your database for different environments. You can look into those files to understand the database operations that SQLAlchemy will perform to apply the migrations. To apply the migrations and create the schemas for these models in the database, run the following command:

```
$ PYTHONPATH=`pwd` alembic heads upgrade
```

This will create the desired schemas in our database. Now that our database models are implemented, and our database contains the desired schemas, it's time to move on to the next step! Go ahead to the next section to learn about the repository pattern!

7.4 Implementing the Repository Pattern for data access

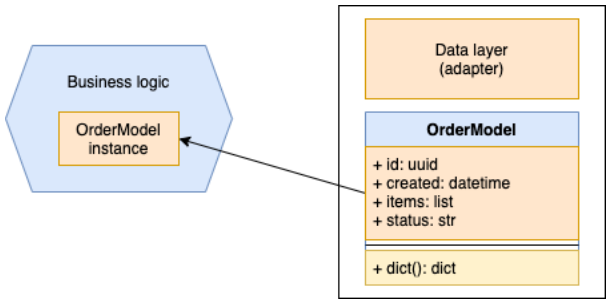
In the previous section, we learned to design the database models for the orders service and to manage changes to the database schema through migrations. With our database models ready, we can interact with the database to create orders and manage them. Now we have to decide how we make the data accessible to the business layer. In this section, we'll first discuss different strategies to connect the business layer with the data layer, and we'll learn what the Repository pattern is, and how we can use it to create an interface between the business layer and the database. Once we understand what the Repository pattern is and how it works, we'll move on to implementing it.

7.4.1 The case for the Repository Pattern: what is it ,and why is it useful?

In this section, we discuss different strategies for interfacing with the database from the business layer, and we introduce the Repository Pattern as strategy that helps us decouple the business layer from the implementation details of the database.

As shown in figure 7.6, a common strategy to enable interactions between the business layer and the database is to use the database models directly within the business layer. Listing 7.2 shows an example of how we can use the `OrderModel` class within the `OrdersService` to create the record for a placed order. Our database models already contain data about the orders, so we could enhance them with methods that implement business capabilities. This is called the Domain Model pattern[5]. This pattern is useful when we have a one-to-one mapping between service capabilities and database operations, or when we don't need the collaboration of multiple domains.

[5] Fowler, p. 116-125.



Figrue 7.6 A common approach to enable interactions between the data layer and the business layer is by using the database models directly in the business layer.

This approach works for simple cases; however, it couples the implementation of the business layer to the database and to the ORM framework of choice. What happens if we want to change the ORM framework later on, or if we want to switch to a different data storage technology that doesn't involve SQL? In those cases, we'd have to make changes to our business layer. This breaks the principles we introduced in section 7.1. Remember, the database is an adapter that the orders service uses to persist data, and the implementation details of the database should not transpire into the business logic. Instead, data access will be encapsulated by our data access layer.

To decouple the business layer from the data layer, we'll use the Repository pattern. This pattern gives us an in-memory list interface of our data. This means that we can get, add, or delete orders from the list, and the repository will take care of translating these operations into database specific commands. Using the Repository Pattern means the data layer exposes a consistent interface to the business layer to interact with the database, regardless of the database technology that we use to store our data. No matter whether we use a SQL database such as PostgreSQL, or a NoSQL database like MongoDB, or an in-memory cache such as Redis, the Repository pattern will remain the same and will use whichever specific operations are required to interact with the database. Figure 7.7 illustrates how the Repository pattern helps us invert the dependency between the data layer and the business layer.

DEFINITION: REPOSITORY PATTERN. Repository is a software development pattern that provides an in-memory list interface to our data store. This helps us decouple our components from the low-level implementation details of the database. The Repository takes care of managing interactions with the database and provides a consistent interface to our components, regarldess of the database technologu used. This allows us to change the database system without having to change our core business logic.

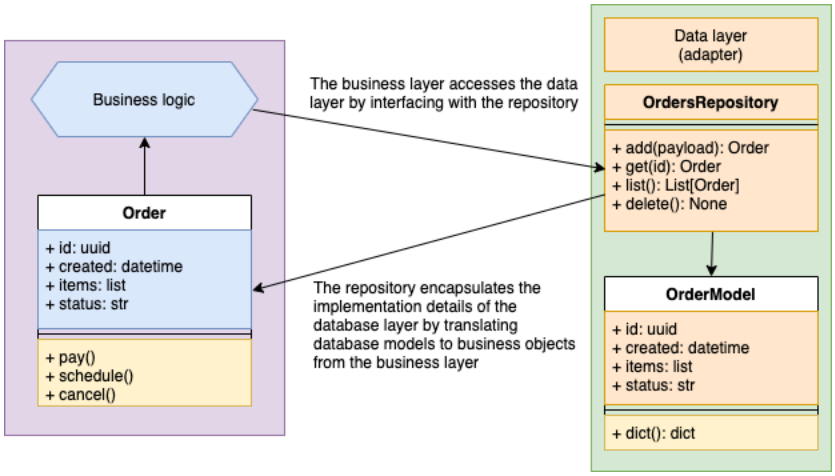


Figure 7.7 The Repository pattern encapsulates the implementation details of the data layer by exposing an in-memory list interface to the business layer, and translates database models to business objects.

Now that we know how we can use the Repository Pattern to allow the business layer to interface with the database while decoupling its implementation from low-level details of the database, we'll learn to implement the Repository Pattern.

7.4.2 Implementing the Repository Pattern

How do we implement the Repository Pattern? We can use different approaches to this as long as we meet the following constraint: none of the operations carried out by the repository can be committed by the repository. What does this mean? It means that when we add an order object to the repository, the repository will add the order to a database session, but it'll not commit the changes. Instead, it'll be the responsibility of the consumer of `OrdersService` (i.e. the API layer) to commit the changes. Figure 7.8 illustrates this process.

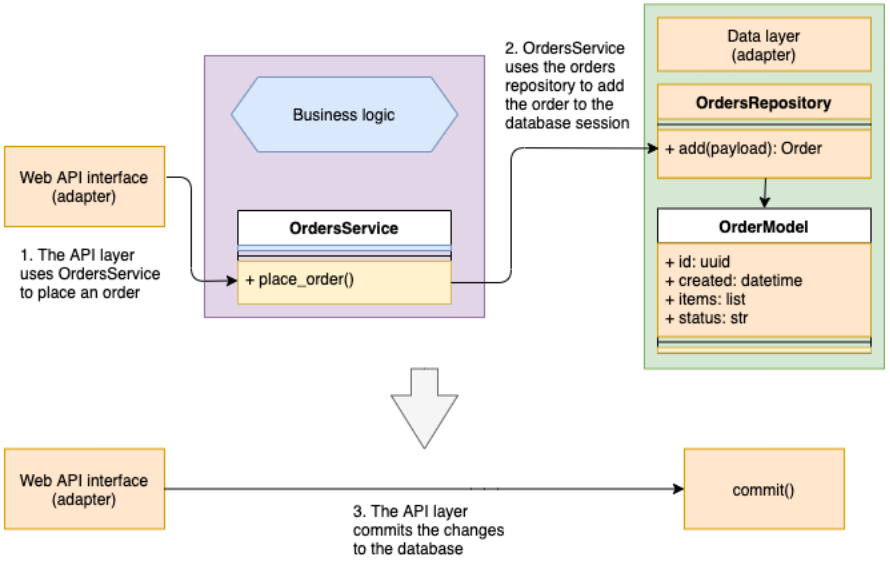


Figure 7.8 Using the Repository pattern, the API layer uses the place_order() capability of OrdersService to place an order. To place the order, OrdersService interfaces with the orders repository to add the order to the database. Finally, the API layer must commit the changes in order to persist them in the database.

Why can't we commit database changes within the repository? First, because the repository acts just like an in-memory list representation of our data, and as such it doesn't have a concept of database sessions and transactions. Second, because the repository is not the right place to execute a database transaction. Instead, the context in which the repository is invoked provides the right context for executing database transactions. In many cases, our applications will execute multiple operations that involve one or more repositories, and also calls to other services. For example, figure 7.9 shows the number of operations involved in processing a payment:

1. The API layer receives the request from the user and uses the `OrdersService`'s `payment_method()` to process the request.
2. `OrdersService` talks to the payments service to process the payment.
3. If the payment is successful, `OrdersService` schedules the order with the kitchen service.
4. `OrdersService` updates the state of the order in the database using the orders repository.
5. If all the previous operations were successful, the API layer commits the transaction in the database, and otherwise it rolls back the changes.

These steps can be taken synchronously, that is one after the other, or asynchronously, that is in no specific order, but regardless of the approach, all steps must succeed or fail all together. As the unit of execution context, it's the responsibility of the API layer to ensure that all changes are committed or rolled back as required. In section 7.6, we'll learn how exactly the API layer controls the database session and commits the transactions.

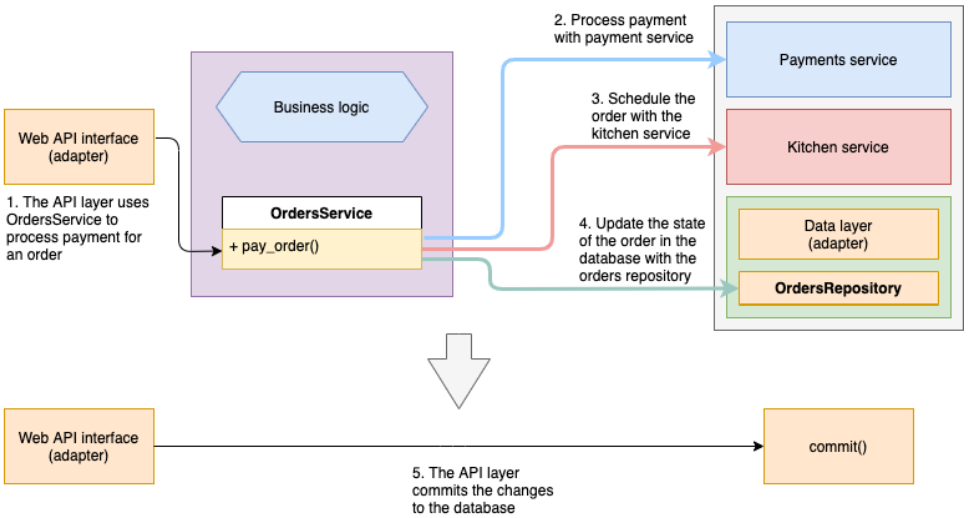


Figure 7.9 In some situations, OrdersService has to interface with multiple repositories or services to perform an operation. In this example, OrdersService interfaces with Payments service to process a payment, then with Kitchen service to schedule the order for production, and finally updates the status of the order through the orders repository. All these operations must succeed or fail together, and it's the responsibility of the API layer to commit or rollback accordingly.

At a minimum, a Repository pattern implementation consists of a class which exposes a `get` and an `add` methods, respectively to be able to retrieve and to add objects to the repository. For the purposes of our implementation, we'll implement also the following methods: `update`, `delete` and `list`. This will simplify the CRUD interface of the repository.

The following question bears some consideration in this context: when we fetch data through the repository, what kind of object should the repository return? In many implementations, you'll see repositories returning instances of the database models (in other words, the classes defined in `orders/repository/models.py`). We won't do that in this chapter. Instead, we'll return objects that represent orders from the business layer domain. Why is it a bad idea to return instances of the database models through the repository? Because it defeats the purpose of the repository, which is to decouple the business layer from the data layer. Remember, we may want to change our persistence storage technology or our ORM framework. If that happens, the database classes we implemented in section 7.2 will no longer exist, and there's no guarantee that a new framework would allow us to return objects with the same interfaces. For this reason, we don't want to couple our business layer with them. Figure 7.8 illustrates the relationship between the business layer and the orders repository.

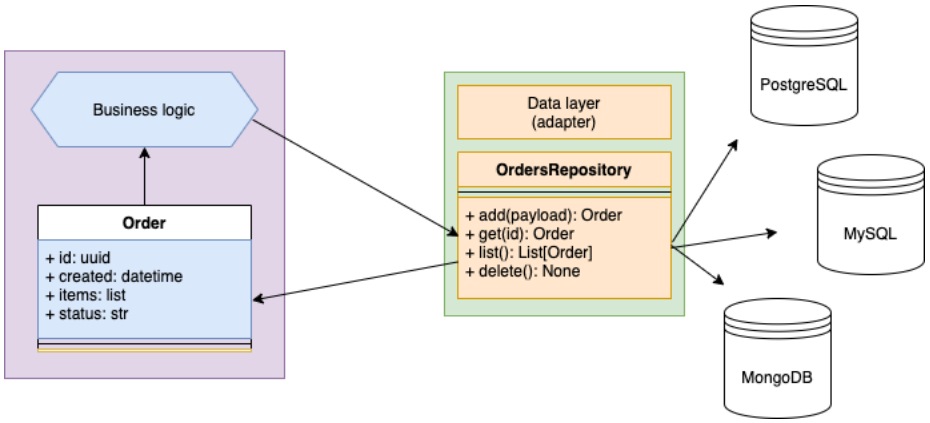


Figure 7.10 The Repository pattern encapsulates the implementation details of the persistent storage technology used to manage our data. Our business layer only ever deals with the repository, and therefore we are free to change our persistent storage solution to a different technology without affecting our core application implementation

Our orders repository implementation will live under `orders/repository/orders_repository.py`. Listing 7.3 shows the implementation of the orders repository. The repository takes one required argument which represents the database session. Objects are added and deleted from the database session. The `add` and `update` methods take a payload that represents an order in the form of a Python dictionary. Our payloads are fairly simple so a dictionary is sufficient here, but if we had more complex payloads, we should consider using objects instead. Except for the `delete` method, all methods of the repository return Order objects from the business layer.

The repository implementation is tightly coupled to the methods of SQLAlchemy's `Session` object, but it also encapsulates these details, and to the business layer the repository appears as an interface to which we submit IDs and payloads, and we get `Order` objects in return. This is the point of the repository: to encapsulate and hide away the implementation details of the data layer from the business layer. This means that if we switch to a different ORM framework, or to a different database system, we'll only need to make changes to the repository.

Listing 7.3 Orders repository

```
from orders.orders_service.orders import Order
from orders.repository.models import OrderModel, OrderItemModel


class OrdersRepository:
    def __init__(self, session):    #A
        self.session = session

    def add(self, items):
        record = OrderModel(items=[OrderItemModel(**item) for item in items])    #B
        self.session.add(record)    #C
        return Order(**record.dict(), order_=record)    #D
```

```
    def _get(self, id_):    #E
        return self.session.query(OrderModel).filter(OrderModel.id == str(id_)).first()    #F

    def get(self, id_):
        order = self._get(id_)    #G
        if order is not None:    #H
            return Order(**order.dict())

    def list(self, limit=None, **filters):    #I
        query = self.session.query(OrderModel)    #J
        if 'cancelled' in filters:    #K
            cancelled = filters.pop('cancelled')
            if cancelled:
                query = query.filter(OrderModel.status == 'cancelled')
            else:
                query = query.filter(OrderModel.status != 'cancelled')
        records = query.filter_by(**filters).limit(limit).all()
        return [Order(**record.dict()) for record in records]    #L

    def update(self, id_, **payload):
        record = self._get(id_)
        if 'items' in payload:    #M
            for item in record.items:
                self.session.delete(item)
            record.items = [
                OrderItemModel(**item) for item in payload.pop('items')
            ]
        for key, value in payload.items():    #N
            setattr(record, key, value)
        return Order(**record.dict())

    def delete(self, id_):
        self.session.delete(self._get(id_))    #O
```

#A To instantiate the repository, we require a session object which we can use to add or delete records from
#B When creating a record for an order, we also need to create a record for each item in the order, which we do in the list comprehension in this line
#C To create a record, we add it to the session object
#D We return an instance of the Order class from business layer (we'll see its implementation soon). To instantiate this class, we use the dictionary returned by the dict() method of our database models
#E Generic method to retrieve a record by ID. The parameter for the ID is named id_ to avoid conflict with the built-in id function of Python
#F We fetch the record using the first() method for SQLAlchemy's query object. first() will return an instance of the record if exists, and otherwise it'll return None. The alternative, using one() would raise an error if the record doesn't exist
#G We use the custom _get() method we defiend ealier to retrieve a record
#H Since the order may not exist, we first check whether _get() returned an order. If _get() returned None, this method will also return None
#I In the list() method, we single out the limit parameter to make it easier to build our query, and we accept any number of additional parameters for filtering
#J We'll build our query dynamically, since different filters will require different query methods
#K If the user wants to filter by whether an order is cancelled or not, we need to add this condition to the query using the filter() method, which allows us to add filters for "equals" and "not equals"
#L We return a list of business objects built with the parameters returned by the dict() method of our database models
#M If a user is updating the items in an order, we'll first delete the items which the order is currently linked to, and create new items. An alternative could be adding a flag to the previous items marking them as invalid, inactive or something similar
#N We update attributes of our database model object dynamically by using the setattr function
#O To delete a record, we call the delete() method from the session object

This completes the implementation of our data layer. We have implemented a persistent storage solution with the help of SQLAlchemy, and we have encapsulated the details of this solution with the help of the repository pattern. It's now time to work on the business layer and see how it will interact with the repository!

7.5 Implementing the business layer

We've done a lot of work designing the database models for the orders service and using the Repository Pattern to build the interface to the data. It's about time to focus on the business layer! In this section, we'll implement the business layer of the orders service. That's the core of the hexagon we introduced in section 7.1 and illustrated in figure 7.1, which is reproduced here as figure 7.11 for your convenience. The business layer implements the service's capabilities. What are the business capabilities of the orders service? From the analysis in section 3.2.2, we know that the orders service is a gateway for the users of the platform to place their orders and manage them.

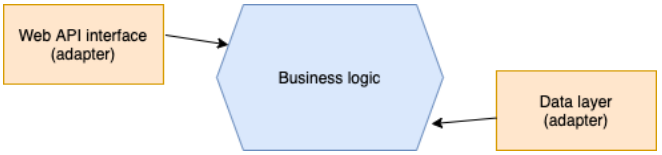


Figure 7.11 In hexagonal architecture, we distinguish a core layer in our application, the business layer, which implements the service's capabilities. Other components, such as a web API interface or a database, are considered adapters which depend on the business layer

As illustrated in figure 7.12, the orders service manages the lifecycle of an order through integrations with other services. The following list describes the capabilities of the orders service, and highlights integrations with other services (refer to figure 7.9 for further clarification):

- **Place orders**: creates a record of an order in the system. The order won't be scheduled in the kitchen until the user pays for it.
- **Process payments**: processes payment for an order with the help of the payments service. If the payments service confirms the payment is successful, the orders service schedules the order for production with the kitchen service.
- **Update orders**: users can update their orders any time to add or remove items from it. To confirm a change, a new payment must be made and processed with the help of the payments service.
- **Cancel orders**: users can cancel their orders anytime. Depending on the status of the order, the orders service will communicate with the kitchen or the delivery service to cancel the order.
- **Schedule order for production in the kitchen**: after payment, the orders service schedules the order for production in the kitchen with the help of the kitchen service.
- **Keep track of orders' progress**: users can keep track of their orders' status through the orders service. Depending on the status of the order, the orders service checks with the kitchen or the delivery service to get updated information about the state of the order.

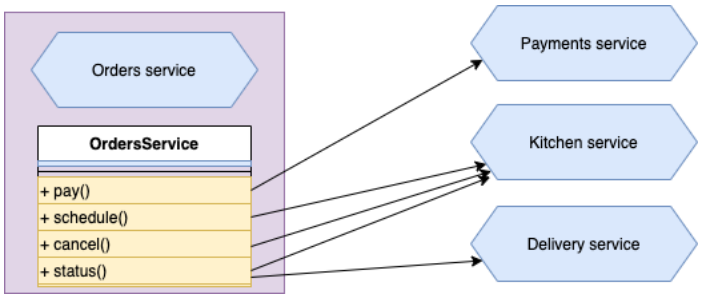


Figure 7.12 In order to perform some of its functions, the orders service needs to interact with orders services. For example, to process payments, it must interact with the payments service, and in order to schedule an order for production, it must interact with the kitchen service

What's the best way to model these actions in our business layer? We can use different approaches to this, but to make it easy for other components to interact with the business layer, we'll expose a single unified interface through a class called `OrdersService`. We'll define this class under `orders/orders_service/orders_service.py`. To fulfill its duties, `OrdersService` uses the orders repository to interface with the database. We could let `OrdersService` import and initialize the orders repository as in the following code snippet:

```
from repository.orders_repository import OrdersRepository

class OrdersService:
    def __init__(self):
        self.repository = OrdersRepository()
```

However, doing this would place too much responsibility on the orders service since it'd need to know how to configure the orders repository. It would also tightly couple the implementation of the orders repository and the orders service, and we wouldn't be able to use different repositories if we needed to. As you can see in figures 7.13 and 7.14, a better approach is to use Dependency Injection in combination with the Inversion of Control principle.

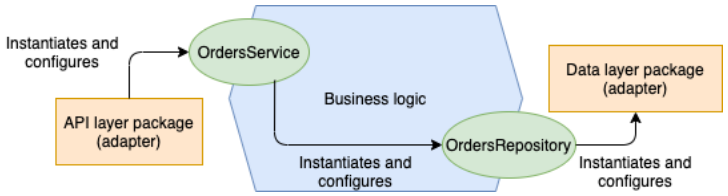


Figure 7.13 In conventional software design, dependencies follow a linear relationship, and each component is responsible for instantiating and configuring its own dependencies. In many cases, this couples our components to low-level implementation details of their dependencies.

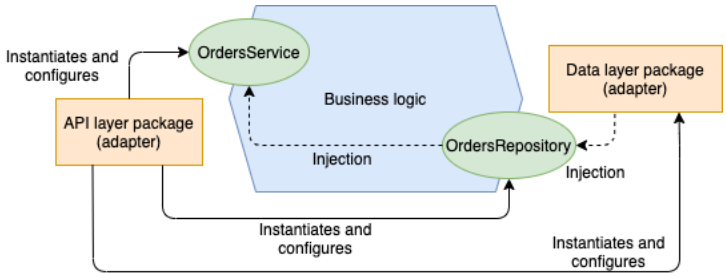


Figure 7.14 With Inversion of Control, we decouple components from their dependencies by supplying them at runtime using methods such as Dependency Injection. In this approach, it's the responsibility of the context to provide correctly configured instances of the dependencies. The solid lines show relationships of dependency, while the dotted lines show how dependencies are injected

> **DEFINITION: INVERSION OF CONTROL.** Inversion of Control is a software development principle that encourages us to decouple our comopnents from their dependencies by supplying them at runtime. This allows us to control how the dependencies are supplied. One popular pattern to accomplish this is Dependency Injection. The context in which the dependencies are instantiated and supplied is called Inversion of Control Container. In the orders service, a suitable Inversion of Control Container is the request object, since most operations are specific to the context of a request.

The inversion of control principle states that we should decouple the dependencies in our code by letting the execution context supply those dependencies at runtime. This means that, instead of letting the orders service import and instantiate the orders repository, we should supply the repository at runtime. How do we do that? We can use different patterns to supply dependencies to our code, but one of the most popular due to its simplicity and effectiveness is dependency injection.

> **DEFINITION: DEPENDENCY INJECTION.** Dependency Injection is a software development pattern whereby we supply code dependencies at runtime. This helps us to decouple our components from the specific implementation details of the code they depend on, since they don't need to know how to congure and instantiate their depdencies.

To make the orders repository injectable into the orders service, we parametrize it:

```
class OrdersService:
    class __init__(self, orders_repository):
        self.orders_repository = orders_repository
```

It's now the responsibility of the caller to instantiate and configure the orders repository correctly. As you can see in figure 7.11, this has a very desirable outcome: depending on the context, we can supply different implementations of the repository or add different configurations. This makes the orders service easier to use in different contexts[6].

[6] For more details on the inversion of control principle and the dependency injection pattern, see Martin Fowler, "Inversion of Control Containers and the Dependency Injection pattern", available at https://martinfowler.com/articles/injection.html [accessed 9th May 2021].

Listing 7.4 shows that interface exposed by the `OrdersService`. The class initializer takes an instance of the orders repository as a parameter to make it injectable. As per the inversion of control principle, when we integrate `OrdersService` with the API layer, it'll be the responsibility of the API to get a valid instance of the orders repository and pass it to `OrdersService`. This approach is convenient, since it allows us to swap repositories at will when necessary, and it'll make it very easy to write our tests in the next chapter.

Listing 7.4 Interface of the OrdersService class

```
class OrdersService:
    def __init__(self, orders_repository):
        self.orders_repository = orders_repository

    def place_order(self, items):
        pass

    def get_order(self, order_id):
        pass

    def update_order(self, order_id, items):
        pass

    def list_orders(self, **filters):
        pass

    def pay_order(self, order_id):
        pass

    def cancel_order(self, order_id):
        pass
```

Some of the actions listed under `OrdersService`, such as payment or scheduling take place at the level of individual orders. Since orders contain data, it'll be useful to have a class that represents orders and has methods to perform tasks related to an order. Within the context of the orders service, an order is a core object of the orders domain. In Domain Driven Design, we call these objects **domain objects**. These are the objects returned by the orders repository. We'll implement our `Order` class under `orders/order_service/orders.py`. Listing 7.5 shows a preliminary implementation of the `Order` class.

In addition to the `Order` class, listing 7.5 also provides an `OrderItem` class to represent each of the items of an order. The `Order` class takes a parameter for each of its attributes. Some of the properties of an order, such as the creation time or its ID, are set by the data layer and can be known only after the changes to the database have been committed. As we explained in section 7.4, committing changes is out of the scope of a repository, which means that, when we add an order to the repository, the returned object won't have those properties yet. For this reason, we allow the Order class to have a pointer to the database record of the order through the `order_` parameter. This is the only degree of coupling we'll allow ourselves between the business layer and the data layer. And to make sure this dependency can be easily removed if we ever have to, we're setting `order_` by default to `None`.

Listing 7.5 Implementation of the Order business object class

```
class OrderItem:    #A
    def __init__(self, id, product, quantity, size):    #B
        self.id = id
        self.product = product
        self.quantity = quantity
        self.size = size


class Order:
    def __init__(self, id, created, items, status, schedule_id=None,
                 delivery_id=None, order_=None):    #C
        self._id = id    #D
        self._created = created
        self.items = [OrderItem(**item) for item in items]    #E
        self._status = status
        self.schedule_id = schedule_id
        self.delivery_id = delivery_id

    @property
    def id(self):    #F
        return self._id or self._order.id

    @property
    def created(self):    #G
        return self._created or self._order.created

    @property
    def status(self):    #H
        return self._status or self._order.status
```

#A Business object representing an order item

#B The method signature for the OrderItem initializer overrides the built-in Python id function. We are allowing ourselves a license here to make it easier to map the parameters in this signature to the values returned by the database model's dict() method. There are no side-effects to this definition in this case

#C The order_ parameter in this method signature represents a database model instance. We don't want to couple our business layer to the specific implementation details of our data layer, so we're setting this parameter by default to None and it's not really required as long as all other parameters are correctly set

#D The value of the parameter id can be undefined when the Order class is instantiated if the database record hasn't been created yet, so we bind the id from the method signature to a provisional self._id attribute, and we'll leave it to the id() property to resolve the actual ID of the order

#E Each item will come in the form of a dictionary, so we want to use their attributes to build OrderItem objects

#F Like we mentioned in #D, the ID of an order is not defined until the database record is created. In those cases, we'll pull the value from the database model instance, which is bound to the self.order_ attribute

#G Same as with the id, the created attribute won't be defined until the record is committed to the database

#H Same as with the id, the status attribute may not be defined until the record is committed to the database

In addition to holding data about an order, the `Order` class also needs to perform certain tasks. We'll encapsulate within this class the tasks of cancelling, paying and scheduling an order. All three tasks involve API calls to other services that need data from the order, so the `Order` class comes as a natural place to encapsulate them. Before we proceed with the implementation, we should know what do those API calls look like.

The folder for this chapter in the GitHub repository contains three OpenAPI files: one for the orders API (oas.yaml), one for the kitchen API (kitchen.yaml), and one for the payments API (payments.yaml). We don't have services implemented for the kitchen and the payments APIs, so

how can we test our integrations with them? We could mock our API calls, but that wouldn't be terribly useful since it doesn't really give us a chance to interact with endpoints and the schemas of the APIs. Since we already have the documentation for those APIs, wouldn't it be amazing if we could somehow run a mock server that replicates the server behind the APIs, validating our requests and returning valid responses? Turns out, we can do that! We'll use a `node.js` Prism CLI (https://github.com/stoplightio/prism), a library built and maintained by Stoplight, to mock the API server for the kitchen and the payments services. Don't worry, it's just a CLI tool, you don't need to know any JavaScript to use it. To install the library, run the following command:

```
$ yarn add prism-cli
```

> **DEALING WITH ERRORS RUNNING PRISM** You may run into errors when running prism. A common error is not having a compatible version of node.js. I recommend you install nvm to manage your node versions, and use the latest stable version of node to run Prism. Also, make sure the port you are selecting to run prism is available.

This command will create a `node_modules` folder within your application folder where prism and all its dependencies will be installed. You don't want to commit this folder! So make sure you add it to your `.gitignore` file. You'll also see a new file called `package.json`, and another one called `yarn.lock` within your application directory. These are the files you want to commit, since they'll allow you to recreate the same `node_modules` directory in any other environment.

To see prism in action with the kitchen API, run the following command:

```
$ ./node_modules/.bin/prism mock kitchen.yaml --port 3000
```

This will start a server on port 3000 running a mock service for the kitchen API. To get a flavor of what we can do with it, run the following command to hit the GET /kitchen/schedule endpoint, which returns a list of schedules:

```
$ curl http://localhost:3000/kitchen/schedule
```

> **DISPLAY JSON IN THE TERMINAL LIKE A PRO WITH JQ** When outputting JSON to the terminal, either using cURL to interact with an API or catting a JSON file, I recommend you use JQ. JQ is a command-line utility that parses the JSON and produces a beautiful display. You can use JQ like this: `curl http://localhost:3000/kitchen/schedule | jq`.

You'll see that the mock server started by prism is able to return a perfectly valid payload representing a list of schedules. Impressive to say the least! Now that know how to run mock servers for the kitchen and the payments APIs, let's analyze the requirements of the API integrations with them:

- Kitchen service (kitchen.yaml): to schedule an order with the kitchen service, we must call the POST /kitchen/schedule endpoint with a payload containing the list of items in the order. In the response to this call, we'll find the `schedule_id`, which we can use to keep track of the state of the order.
- Payments service (payments.yaml): to process the payment for an order, we must call the POST /payments endpoint with a payload containing the ID of the order. This is a mock endpoint for integration testing purposes.

Also, to cancel an order, we must check whether the status of the order. If the order is scheduled for production, we must hit the POST `/kitchen/schedule/{schedule_id}/cancel` endpoint to cancel the schedule. If the order is out for delivery, we don't allow users to cancel the order and therefore we raise an exception.

To implement the API integrations, we'll use the popular Python requests library. Run the following command to install the library with Pipenv:

```
$ pipenv install requests
```

Listing 7.6 extends the implementation of the `Order` class by adding methods that implement API calls to the kitchen and the payment services (the new code is shown in bold case). For testing purposes, we're expecting the kitchen API to run on port 3001 and the payments service to run on port 3000. You can accomplish this by running the following commands:

```
$ ./node_modules/.bin/prism mock payments.yaml --port 3000
$ ./node_modules/.bin/prism mock kitchen.yaml --port 3001
```

In each API call, we check that the response contains the expected status code, and if it doesn't, we raise a custom `APIIntegrationError` exception. Also, if a user tries to perform an invalid action, such as cancelling an order when it's already out for delivery, we raise an `InvalidActionError` exception.

Listing 7.6 Encapsulating per-order capabilities within the Order class

```
import requests

from orders.orders_service.exceptions import (
    APIIntegrationError, InvalidActionError
)


...
class Order:
  ...

    def cancel(self):
        if self.status == 'progress':    #A
            response = requests.post(
                f'http://localhost:3001/kitchen/schedule/{self.schedule_id}/cancel',
                data={'order': self.items}
            )
            if response.status_code == 200:    #B
                return
            raise APIIntegrationError(    #C
                f'Could not cancel order with id {self.id}'
            )
        if self.status == 'delivery':    #D
            raise InvalidActionError(f'Cannot cancel order with id {self.id}')

    def pay(self):
        response = requests.post(    #E
            'http://localhost:3000/payments', data={'order_id': self.id}
        )
        if response.status_code == 200:
            return
```

```
        raise APIIntegrationError(
            f'Could not process payment for order with id {self.id}'
        )

    def schedule(self):
        response = requests.post(    #F
            'http://localhost:3001/kitchen/schedule',
            data={'order': self.items}
        )
        if response.status_code == 201:    #G
            return response.json()['id']
        raise APIIntegrationError(
            f'Could not schedule order with id {self.id}'
        )
```

#A If an order is in progress, it means it's being produced in the kitchen, so to cancel it we need to cancel its schedule by making an API call to the kitchen service
#B If the response from the kitchen service is successful, we simply return
#C Otherwise we raise an APIIntegrationError
#D We don't allow users to cancel their orders which are being delivered, so if the status is set to 'delivery' we raise an error
#E To process a payment, we need to make an API call to the payments service
#F To schedule an order for production, we need to make an API call to the kitchen service
#G If the response from the kitchen service is successful, we return the ID from the response payload, which we'll use to set the schedule_id of the order

Listing 7.7 contains the implementation of the custom exceptions we use in the order service to signal that something has gone wrong. We'll use `OrderNotFoundError` in the `OrdersService` class when a user tries to fetch the details of an order that doesn't exist.

Listing 7.7 Orders service custom exceptions

```
class OrderNotFoundError(Exception):    #A
    pass


class APIIntegrationError(Exception):    #B
    pass


class InvalidActionError(Exception):    #C
    pass
```

#A Exception to signal that an order doesn't exist
#B Exception to signal that an API integration error has taken place
#C Exception to signal that the action being performed is invalid

Now, as we mentioned earlier, the API module won't be using the Order class directly. Instead, we'll use a unified interface to all our adapters through the `OrdersService` class, whose interface we showed in listing 7.8. `OrdersService` encapsulates the capabilities of the orders domain, and it takes care of using the orders repository to get orders objects and perform actions on them. Listing 7.8 shows the implementation of the `OrdersService` class.

Listing 7.8 Implementation of the OrdersService

```
from orders.orders_service.exceptions import OrderNotFoundError


class OrdersService:
    def __init__(self, orders_repository):    #A
        self.orders_repository = orders_repository

    def place_order(self, items):
        return self.orders_repository.add(items)    #B

    def get_order(self, order_id):
        order = self.orders_repository.get(order_id)    #C
        if order is not None:    #D
            return order
        raise OrderNotFoundError(f'Order with id {order_id} not found')

    def update_order(self, order_id, items):
        order = self.orders_repository.get(order_id)
        if order is None:
            raise OrderNotFoundError(f'Order with id {order_id} not found')
        return self.orders_repository.update(order_id, {'items': items})

    def list_orders(self, **filters):
        limit = filters.pop('limit', None)    #E
        return self.orders_repository.list(limit, **filters)

    def pay_order(self, order_id):
        order = self.orders_repository.get(order_id)
        if order is None:
            raise OrderNotFoundError(f'Order with id {order_id} not found')
        order.pay()
        schedule_id = order.schedule()    #F
        return self.orders_repository.update(
            order_id, {'status': 'scheduled', 'schedule_id': schedule_id}
        )

    def cancel_order(self, order_id):
        order = self.orders_repository.get(order_id)
        if order is None:
            raise OrderNotFoundError(f'Order with id {order_id} not found')
        order.cancel()
        return self.orders_repository.update(order_id, {'status': 'cancelled'})
```

#A To instantiate the OrdersService class, we require an orders repository object that we can use to add or delete orders from our records

#B To place an order we need to create a record in the database, and to do that we simply add the details of the order to the repository

#C To get the details of an order, we ask the repository to give us an order with the requested ID

#D If the requested order doesn't exist, the repository returns None, so we need to check the return value of this operation. If the order doesn't exist, we raise an OrderNotFoundError

#E Filters come in the form of a dictionary. To get a list of orders, the orders repository forces us to pass a specific value for the limit filter. If the value is None, the repository won't cut the list of results. We extract the value for limit from the filters dictionary by using the pop method, which also removes the key from the dictionary. We also provide a default value of None for limit in case it hasn't been set

#F When we schedule an order with the kitchen service, we get the ID of the schedule, which we use to set the schedule_id attribute of the order

The orders service is ready to be used in our API module. However, before we get on with this integration, there's one more piece in this puzzle that we need to solve. As we mentioned in section 7.4, the repository orders doesn't commit any actions to the database. It's the responsibility of the API, as the consumer of the `OrdersService`, to ensure that everything is committed at the end of an operation. How exactly does that work? Move on to the next section to learn how!

7.6 Implementing the Unit of Work pattern

In this section, we'll learn how to use how to handle database commits and rollbacks when interacting with the `OrdersService`. As you can see in figure 7.10, when we use the `OrdersService` class to access any of its capabilities, we must inject an instance of the `OrdersRepository` class. We must also open a SQLAlchemy session before we perform any actions, and we must commit any changes to our data to persist them in the database.

What's the best way to orchestrate these operations? We can use different approaches to this implementation. As you can see in figure 7.15, we could simply use SQLAlchemy session objects to wrap our calls to the `OrdersService`, and once our operations succeed use the session to commit, or rollback otherwise. This would work if the `OrdersService` only ever had to deal with a single SQL database. However, what if we had to interact with a different type of database at the same time? Sure enough, we'd open a new session for it as well. What if we also had handle integrations with other microservices within the same operation, and ensure we make the right API calls at the end of the transaction in case we had to roll back? Again, we could just add special clauses and guards to our code. The same code would have to be repeated in every API function that interacts with the `OrdersService`, so wouldn't it be nice if there was pattern that can help us put it all together in a single place? Enter the Unit of Work pattern.

> **DEFINITION: UNIT OF WORK PATTERN.** Unit of Work is a design pattern that guarantees the atomicity of our business transactions, ensuring that all transactions are committed at once, or rolled back if any of them fails.

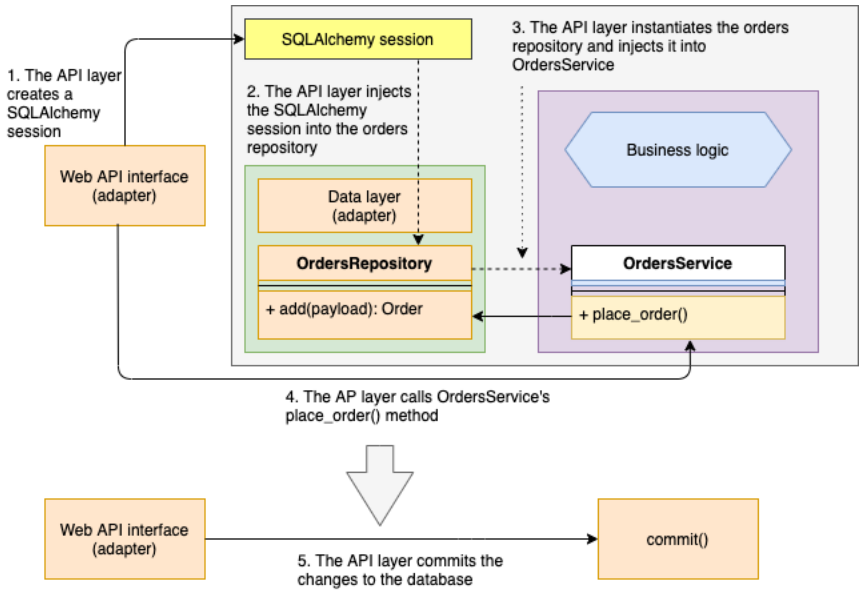


Figure 7.15 To persist our changes to the database, we could simply make the API layer use the SQLAlchemy session object to commit the transaction. In this figure, the solid lines represent calls, while the dashed lines represent injection of dependencies.

Unit of Work is a pattern that ensures that all objects of a business transaction are changed together, and if something fails, it ensures none of them changes[7]. The notion comes from the world of databases, where database transactions are implemented as units of work which ensure that every transaction is

- atomic: the whole transaction either succeeds or fails,
- consistent: it conforms to the constrains of the database,
- isolated: it doesn't interfere with other transactions, and
- durable: it's written to persistent storage

These properties are known as the ACID principles in the world of databases (https://en.wikipedia.org/wiki/Database_transaction). When it comes to services, the Unit of Work pattern helps us to apply these principles in our operations. SQLAlchemy's `Session` object already implements the Unit of Work pattern for database transactions[8]. This means that we can add as many changes as we need to the same session and commit them all together. If something goes wrong, we can call the `rollback` method to undo any changes. In Python, we can orchestrate these steps with context managers.

As you can see in figure 7.16, a context manager is a pattern that allows us to lock a resource during an operation, ensure that any necessary cleanup jobs are undertaken in case anything goes wrong, and finally release the lock once the operation is finished. The key syntactical feature of a

[7] Martin Fowler, *Patterns of Enterprise Architecture*, pp. 184-194.
[8] https://docs.sqlalchemy.org/en/13/orm/session_basics.html

context manager is that use it with the `with` statement, as illustrated in figure 7.16 As you can see in the illustration, context managers can return objects which we can capture by using an `as` clause. This is useful if the context manager is creating access to a resource, such as a file, on which we want to operate.

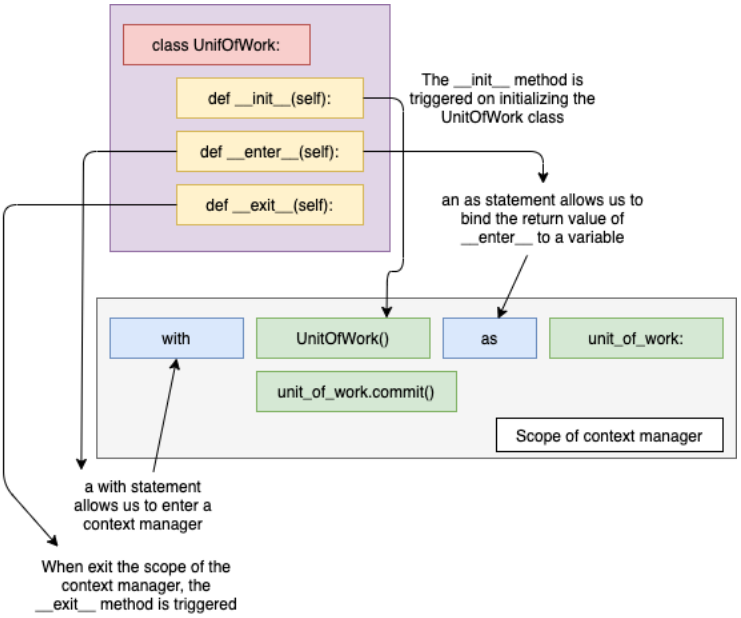


Figure 7.16 A class-based context manager has an __init__, an __enter__, and an __exit__ methods. __init__ is triggered when we initialize the context manager. The __enter__ method allows us to enter the context, and it's called when we use the "with" statement. Using an "as" statement within the same line allows us to bind the return value of the __enter__ metrhod to a variable (unit_of_work in this case). Finally, when we exit the context manager, the __exit__ method is triggered.

In Python, we can implement context managers in multiple ways, including as a class or using the `@contextmanager` decorator from the `contextlib` module[9]. In this section, we'll implement our Unit of Work context manager as a class. A context manager class must implement at least the two following special methods:

- `__enter__`: defines the operations that must be undertaken on entering the context, such as creating a session or opening a file. If we need to perform actions on any of the objects created within the `__enter__` method, we can return the object and capture its value through an as clause, as illustrated in figure 7.16.
- `__exit__`: defines the operations that must be undertaken on exiting the context, for example closing a file or a session. The `__exit__` method captures any exceptions raised during the execution of the context through three parameters in its method signature:

[9] Ramalho, Fluent Python, pp. 463-478.

- `exc_type`: captures the type of exception raised.
- `exc_value`: captures the value bound to the exception, typically the error message.
- `traceback`: a traceback object that can be used to pinpoint to the exact place where the exception took place.

If no exceptions are raised, the value of these three parameters will be `None`.

Listing 7.9 shows the implementation of the Unit of Work pattern as a context manager for the orders service. When we enter the context, we create a new database session, and we return the context object itself. The database session is accessible as an attribute of the context object. On existing the context, we first check whether any errors were raised while adding or removing objects to the session, and if that's the case, we rollback the changes. If no errors were raised, we commit all changes.

Listing 7.9 Unit of Work pattern as a context manager

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker


class UnitOfWork:

    def __init__(self):
        self.session_maker = sessionmaker(    #A
            bind=create_engine('sqlite:///orders.db')
        )

    def __enter__(self):
        self.session = self.session_maker()    #B
        return self    #C

    def __exit__(self, exc_type, exc_val, traceback):    #D
        if exc_type is not None:    #E
            self.rollback()    #F
            self.session.close()    #G
        self.session.close()    #H

    def commit(self):
        self.session.commit()    #I

    def rollback(self):
        self.session.rollback()    #J
```

#A At the time of instantiating the Unit of Work, we get a pointer to the session_maker factory function of SQLAlchemy. In this case, we are hardcoding the database connection string to our local SQLite database

#B On entering the Unit of Work context, we get an instance of SQLAlchemy's session object

#C We return an instance of the Unit of Work object so that the caller can access any of its attributes, such as the session object or the commit method

#D On existing the context, we have access to any exceptions that occurred during the execution of the context. If an exception was raised, we have access to its type, its value, and the traceback context. If no exception took place, all three parameters will be set to None

#E We check whether an exception took place by checking whether exc_type is None

#F If an exception took place, we must roll back all the transactions accumulated in the session object to avoid leaving the database in an inconsistent state

#G On existing the context, we must close the session to free up resources in the database

#H If no exception took place, we simply close the session and exit
#I Wrapper around the commit() method from SQLAlchemy's session object. If we were working with more than one database, we could handle all commits within this method
#J Wrapper around the rollback() method from SQLAlchemy's session object. If we were working with more than one database, we could handle all rollbacks within this method

This is all very good, but how exactly are we supposed to use the `UnitOfWork` in combination with the orders repository and the `OrdersService`, I hear you ask? In the next section, we'll delve more into the details of this, but before we do that, listing 7.10 gives you a template pattern for how to use all these components together.

Listing 7.10 Template pattern for using the Unit of Work and the Repository

```
with UnitOfWork() as unit_of_work:    #A
    repo = OrdersRepository(unit_of_work.session)    #B
    orders_service = OrdersService(repo)    #C
    orders_service.place_order(order_details)    #D
    unit_of_work.commit()    #E
```

#A We enter the Unit of Work context with the syntax of context managers, using a 'with' statement. We also use an 'as' statement to bind the return value of UnitOfWork's __enter__ method to the unit_of_work variable
#B We get an instance of the orders repository passing in SQLAlchemy's session object from the UnitOfWork's object
#C We get an instance of the OrdersService class passing in the orders repository object
#D We use the orders service object to place an order
#E We commit the changes to the database

Now that we have a unit of work that we can use to commit our transactions, let's see how we put this all together by integrating the API layer with the service layer! Move on to the next section to learn how we do that!

7.7 Integrating the API layer and the service layer

In this section, we put everything we have learned in this chapter together to integrate the service layer with the API layer. We'll make use of the template pattern we showed in listing 7.10 to use the UnitOfwork class in combination with OrdersRepository and OrderService. When a user is trying to perform an action on an order, we'll make sure we have checks in place to verify that the order exists in the first place, and otherwise we'll return a 404 (Not Found) error response.

Listing 7.11 shows the new version of the `orders/web/api/api.py` module. The first thing we do in every function is to enter the context of `UnitOfWork`, making sure we bind the context object to a variable, `unit_of_work`. Then we create an instance of `OrdersRepository` using the session object from the `UnitOfWork` context object. Once we have an instance of the repository, we inject it into `OrdersService` as we create an instance of the service. Then we use the service to perform the operations required in each endpoint. In endpoints that perform actions on a specific order, we guard against the possibility of an `OrderNotFoundError` being raised by `OrderService` if the requested order doesn't exist.

Notice that, in the `create_order` function, we retrieve the dictionary representation of the order using `order.dict()` before we exit the `UnitOfWork` context. We do it this way because, we need to go back to the implementation of the `Order` class in `orders/orders_service/orders.py`. Remember that the order ID doesn't exist until the changes are committed to the database, so in

the `id` property of the `Order` class we check whether `self._id` is set, and if it's not, we access the ID from the `self._order` object. `self._order` is a pointer to the SQLAlchemy model object (i.e. the `OrderModel` class from `orders/repository/models.py`), and when creating an order record in the database, SQLAlchemy populates the ID with value issued from the database. The trick is, the properties of the `OrderModel` object are only accessible as long as the SQLAlchemy session is active. In our implementation, that means that we have to access the ID before we exit the `UnitOfWork` context, since the database session is closed right before exiting the context. Figure 7.17 illustrates this process.

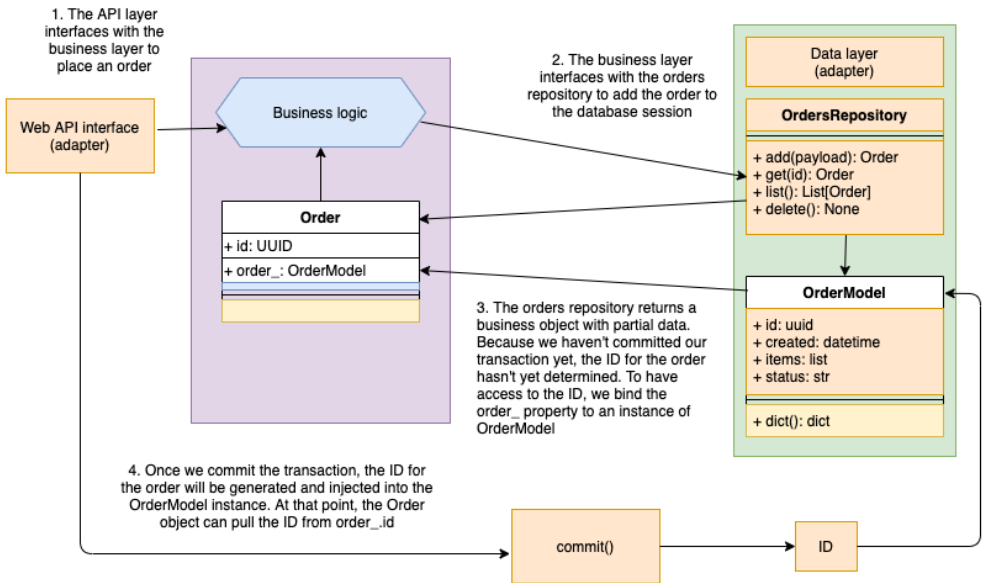


Figure 7.17 When we place an order, the object returned by the orders repository doesn't yet contain an ID. The ID will be available once we commit the database transaction through the OrderModel instance. Therefore, we bind an instanced of the model to the Order object so that it can pull the ID from the model after the commit

Listing 7.11 Integration between API layer and service layer

```
from http import HTTPStatus
from typing import List, Optional
from uuid import UUID

from fastapi import HTTPException
from starlette import status
from starlette.responses import Response

from orders.orders_service.exceptions import OrderNotFoundError
from orders.orders_service.orders_service import OrdersService
from orders.repository.orders_repository import OrdersRepository
from orders.repository.unit_of_work import UnitOfWork
from orders.web.app import app
```

```
from orders.web.api.schemas import GetOrderSchema, CreateOrderSchema


@app.get('/orders', response_model=List[GetOrderSchema])
def get_orders(cancelled: Optional[bool] = None, limit: Optional[int] = None):
    with UnitOfWork() as unit_of_work:    #A
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        results = orders_service.list_orders(limit=limit, cancelled=cancelled)
    return [result.dict() for result in results]


@app.post('/orders', status_code=status.HTTP_201_CREATED, response_model=GetOrderSchema)
def create_order(payload: CreateOrderSchema):
    with UnitOfWork() as unit_of_work:
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        order = orders_service.place_order(payload.dict()['order'])    #B
        unit_of_work.commit()
        return_payload = order.dict()    #C
    return return_payload


@app.get('/orders/{order_id}', response_model=GetOrderSchema)
def get_order(order_id: UUID):
    try:    #D
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.get_order(order_id=order_id)
        return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )


@app.put('/orders/{order_id}', response_model=GetOrderSchema)
def update_order(order_id: UUID, order_details: CreateOrderSchema):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.update_order(
                order_id=order_id, items=order_details.dict()['order']
            )
            unit_of_work.commit()
        return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )


@app.delete('/orders/{order_id}', status_code=status.HTTP_204_NO_CONTENT)
def delete_order(order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
```

```
            orders_service = OrdersService(repo)
            orders_service.delete_order(order_id=order_id)
            unit_of_work.commit()
        return Response(status_code=HTTPStatus.NO_CONTENT.value)
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )


@app.post('/orders/{order_id}/cancel', response_model=GetOrderSchema)
def cancel_order(order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.update_order(
                order_id=order_id, status='cancelled'
            )
            unit_of_work.commit()
        return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )


@app.post('/orders/{order_id}/pay', response_model=GetOrderSchema)
def pay_order(order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.pay_order(order_id=order_id)
            unit_of_work.commit()
        return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )
```

#A Getting a list of orders follows the same patterns we showed in listing 7.10 for using the Unit of Work pattern in combination with the Repository pattern

#B To place an order, we fetch the details of the order in dictionary format using the dict() method which Pydantic schemas conveniently offer for this operation

#C To get all of the details of the newly created order, we must fetch them using the dict() method right after committing the changes and before exiting the context. Remember, on exiting the context, we close SQLALchemy's database session, and once the session is closed, our database model objects don't have access to the values generated by the database, such as the order ID

#D In this case, we wrap our operation within a try/except block to catch the OrderNotFoundError, which will be raised if the requested order doesn't exist. If the error is raised, we return a 404 error response to our users

This concludes our journey through the implementation of the service layer for the orders service. The patterns we have learned in this chapter are not only applicable to the world of APIs and microservices, but to all application models generally. In particular, the Repository pattern will always help you ensure that you keep your data access layer is fully decoupled from the business

layer, and the Unit of Work pattern will help you ensure that all transactions of a business operation are handled atomically and consistently.

7.8 Summary

- Hexagonal architecture, or architecture of ports and adapters, is a software architectural pattern that encourages us to decouple the business layer from the implementation details of the database and the application interface.
- The Dependency Inversion Principle teaches us that the implementation details of our application components should depend on interfaces. This helps us to decouple our components from the implementation details of their dependencies.
- To interface with the database, you can use an Object Relational Mapper library such as SQLAlchemy, which can translate database tables and rows into classes and objects. This gives us the possibility to enhance our database models with useful functionality for our application needs.
- Repository is a software development pattern that helps to decouple the data layer from the business layer by adding an abstraction layer which exposes an in-memory list interface of the data. Regardless of the database engine that we use, the business layer will always receive the same objects from the repository.
- The Unit of Work pattern helps us to ensure that all the business transactions which are part of an application operation succeed or fail together. If one of the transactions fails, the Unit of Work pattern ensures that all changes are rolled back. This mechanism ensures that our data is never left in an inconsistent state.

8

Designing GraphQL APIs

This chapter covers

- Understanding what GraphQL works
- Producing an API specification using the Schema Definition Language (SDL)
- Learning GraphQL's built-in scalar types and data structures and building custom object types
- Creating meaningful connections between GraphQL types
- Designing GraphQL queries and mutations

GraphQL is one of the most popular protocols for building web APIs. It's a suitable choice for driving integrations between microservices, and for building integrations with frontend applications. GraphQL gives API consumers full control over the data that they want to fetch from the server, and how they want to fetch it.

In this chapter, you'll learn to design a GraphQL API. You'll do it by working on a practical example: you'll design a GraphQL API for the products service of the CoffeeMesh platform. The products service owns data about the products offered by CoffeeMesh as well as their ingredients. Each product and each ingredient contains a rich list of properties that describe their features. However, when a client requests a list of products, they most likely interested in fetching only a few details about each product. Also, clients may be interested in being able to traverse the relationships between products, ingredients, and other objects owned by the products service. For these reasons, GraphQL is an excellent choice for building the products API.

As we build the specification for the products API, you'll learn about GraphQL's scalar types, designing custom object types, as well as queries and mutations. By the end of this chapter, you'll understand how GraphQL compares with other types of APIs and when it makes most sense to use it. We've got a lot to cover, so without further ado, let's start our journey!

To follow along with the specification we develop in this chapter, you can use the GitHub repository provided with this book. The code for this chapter is available under the folder named ch09.

8.1 Introducing GraphQL

This section covers what GraphQL is, what its advantages are, and when it makes more sense to use it. The official website of the GraphQL specification defines GraphQL as a "query language for APIs and a runtime for fulfilling those queries with your existing data."[1] What does this really mean? It means that GraphQL is a specification that allows us to run queries in an API server. In the same way SQL provides a query language for databases, GraphqL provides a query language for APIs[2]. GraphQL also provides a specification for how those queries are resolved in a server, o that anyone can implement a GraphQL runtime in any programming language.[3]

Just as we can use SQL to define schemas for our database tables, so we can use GraphQL to write specifications that describe the type of data that can be queried from our servers. A GraphQL API specification is called a **schema**, and it's written in a standard called Schema Definition Language (SDL). In this chapter, we're going to learn how to use the Schema Definition Language to produce a specification for the products API.

GraphQL was first released in 2015, and since then it's been gaining traction as one of the most popular choices for building web APIs. I should say there's nothing in the GraphQL specification saying that GraphQL should be served over HTTP, but in practice, this is the most common type of protocol used in GraphQL APIs.

What's great about GraphQL? GraphQL shines in giving users full control over which data they want to obtain from the server. For example, as we'll see in the next section, in the products API we store many details about each product, such as its name, price, availability, and ingredients, among others. As you can see in figure 8.1, if a user wishes to get a list of products including just their names and prices, with GraphQL they can do that. In contrast, with other types of APIs, such as REST, you'd get a full list of details for each product. Therefore, whenever it's important to give the client full control over how they fetch data from the server, GraphQL is a great choice.

[1] This definition appears in the homepage of the GraphQL specification: https://graphql.org/ [accessed 2021/07/01].
[2] I owe the comparison between GraphQL and SQL to Eve Porcello and Alex Banks, *Learning GraphQL, Declarative Data Fetching for Modern Web Apps*, Sebastopol (CA), O'Reilly, 2018, pp. 31-32.
[3] The GraphQL website maintains a list of runtimes available for building GraphQL servers in different languages: https://graphql.org/code/ [accessed 2021/07/01].

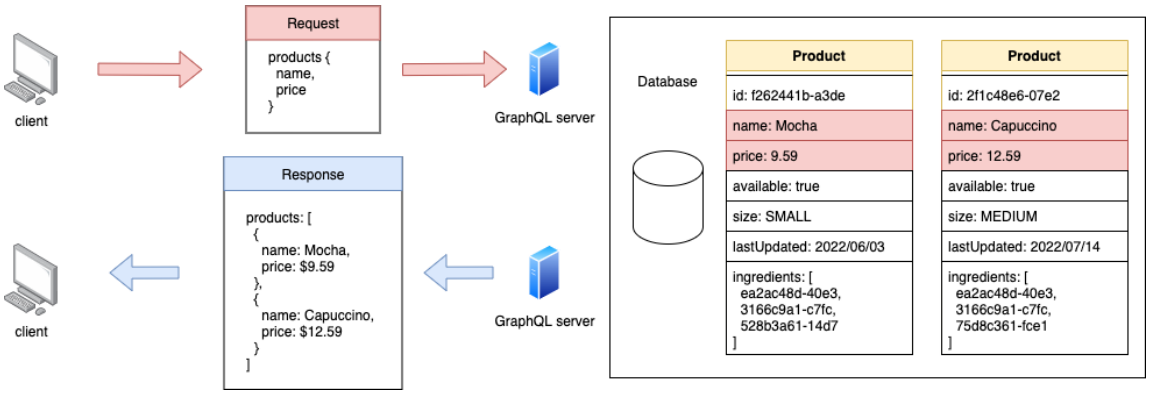


Figure 8.1 Using a GraphQL API, a client can request a list of items with specific details. In this example, a client is requesting the name and price of each product in the products API.

Another great advantage of GraphQL is the ability to create connections between different types of resources, and to expose those connections to our clients for use in their queries. For example, in the products API, products and ingredients are different but related types of resources. As you can see in figure 8.2, if a user wants to get a list of products, including their names, prices, and the names of their ingredients, with GraphQL they can do that by leveraging the connections between these resources. Therefore, in services where we have highly interconnected resources, and where it's useful for our clients to explore and query those connections, GraphQL makes an excellent choice.

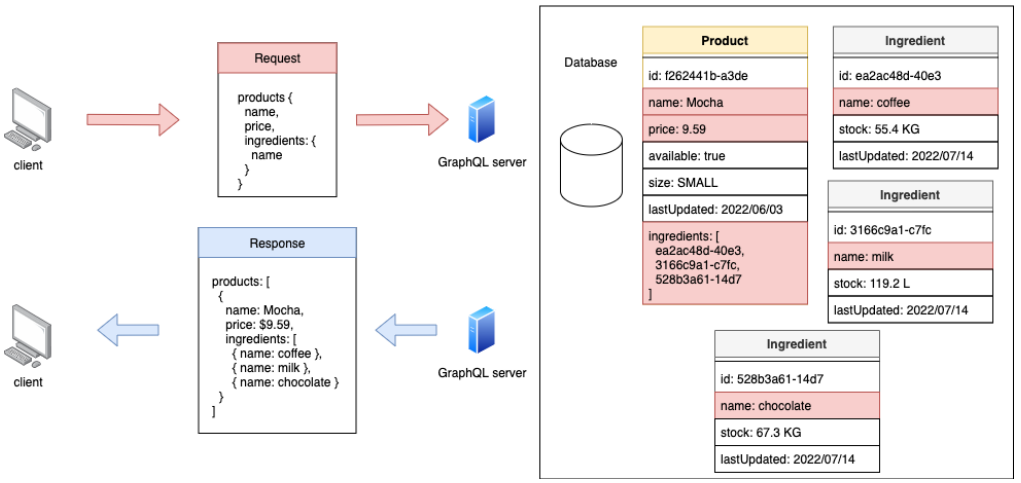


Figure 8.2 Using GraphQL, a client can request the details of a resource and other resources linked to it. In this example, the products API has two types of resources: products and ingredients, both of which are connected through product's ingredients field. Using this connection, a client can request the name and price of each product, as well as the name of each product's ingredient.

In the sections that follow, we'll learn how to produce a GraphQL specification for the products service. We'll learn how to define the types of our data, how to create meaningful connections between resources, and how to define operations for querying the data and changing the state of the server. But before we do that, we ought to understand what the requirements for the products API are, and that's what we do in the next section!

8.2 Introducing the products API

This section discusses the requirements of the products API. Before working on an API specification, it's important to gather information about the API requirements, and that's what we do in this section. As you can see in figure 8.3, the products API is the interface to the products service. To determine the requirements of the products API, we need to know what users of the products service must be able to do with it.

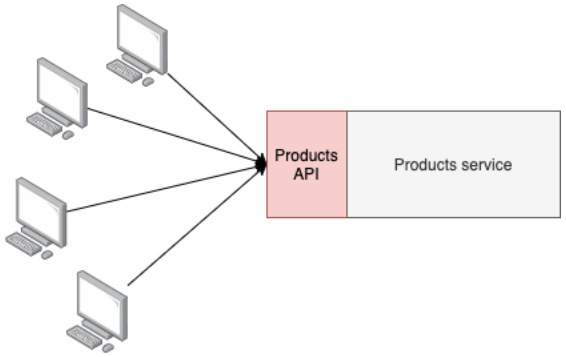


Figure 8.3 To interact with the products service, clients use the products API.

The products service owns data about the products offered by the CoffeeMesh platform. As you can see in figure 8.4, the CoffeMesh staff must be able to use the products service to manage the available stock of each product, as well as to keep the products' ingredients up to date. In particular, they must be able to query the stock of a product or ingredient, and to update them when new stock arrives to the warehouse. They must also be able to add new products or ingredients to the system, and to delete old ones. This information already gives us a complex list of requirements, so let's break it down into specific technical requirements.

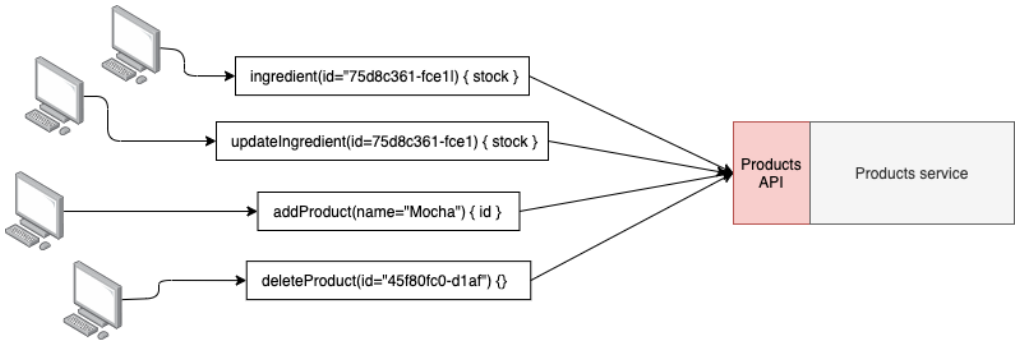


Figure 8.4 To manage products and ingredients, the CoffeeMesh staff uses the products service.

Let's start with by modeling the resources managed by the products API. We want to know which type of resources we should expose through the API, and what are the products' properties. From the description in the previous paragraph, we know that the products service manages two types of resources: products and ingredients. Let's analyze products first.

The CoffeeMesh platform offers two types of products: cakes and beverages. As you can see in figure 8.5, both cakes and beverages have a common set of properties, including the product's name, price, size, list of ingredients, and their availability. Cakes have two additional properties:

- hasFilling, which indicates whether the cake has a filling.
- hasNutsToppingOption, which indicates whether the customer can add a topping of nuts to the cake.

Meanwhile, beverages have the following two additional properties:

- hasCreamOnTopOption, which indicates whether the customer can top the beverage with cream.
- hasServeOnIceOption, which indicates whether the customer can choose to get the beverage served on ice.

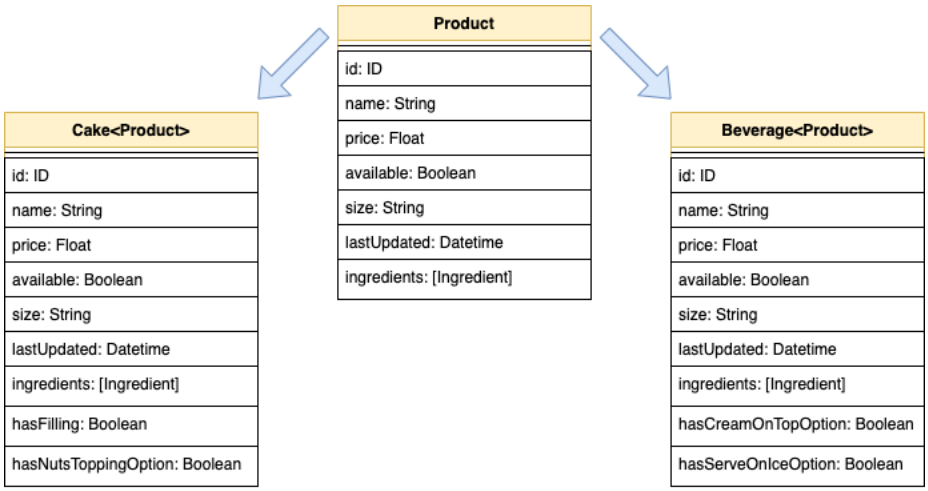


Figure 8.5 CoffeeMesh exposes two types of products: Cake and Beverage, both of which share a common list of properties.

What about ingredients? As you can see in figure 8.6, we can represent all ingredients through one entity with the following attributes:

- name: the ingredient's name.
- stock: the ingredient's available stock. Since different ingredients are measured with different units, such as kilograms or liters, we express the available stock in terms of amounts of per unit of measure.
- description: a collection of notes that CoffeeMesh employees can use to describe and qualify the product.
- supplier: information about the company that supplies the ingredient to CoffeMesh, including their name, address, contact number, and email.

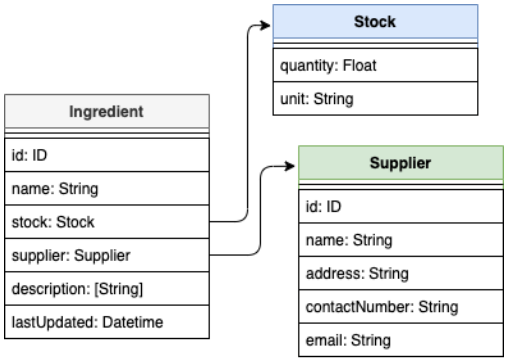


Figure 8.6 List of properties that describe an ingredient. The ingredient's supplier is described by a resource called Supplier, while the ingredient's stock is described through a Stock object.

Now that we've modelled the main resources managed by the products service, let's turn our attention to the operations we must expose through the API. We'll distinguish read operations from write/delete operations. This distinction will make sense when we look more closely at these operations in sections 8.8 and 8.9.

Based on the previous discussion, we'll expose the following read operations:

- `allProducts()` returns the full list of products available in the CoffeeMesh catalogue.
- `allIngredients()` returns the full list of ingredients used by CoffeeMesh to make their products.
- `products()` allows users to filter the full list of products by certain criteria such as availability, maximum price, and others.
- `product()` allows users to obtain information about a single product.
- `ingredient()` allows users to obtain information about a single ingredient.

In terms of write/delete operations, from the previous discussion it's clear that we should expose the following capabilities:

- `addIngredient()` to add new ingredients.
- `updateStock()` to update an ingredient's stock.
- `addProduct()` to add new products.
- `updateProduct()` to update existing products.
- `deleteProduct()` to delete products from the catalogue.

Now that we understand the requirements of the products API, it's time to move on to creating the API specification! In the following sections, we'll learn to create a GraphQL specification for the products API, and along the way, we'll learn how GraphQL works. Our first stop is GraphQL's type system, which we'll use to model the resources managed by the APIs. Move on to the next section to learn more about this!

8.3 Introducing GraphQL's type system

In this section, we introduce GraphQL's type system. In GraphQL, types are definitions that allow us to describe the properties of our data. They're the building blocks of a GraphQL API, and we use them to model the resources owned by the API. In this section, you'll learn to use GraphQL's type system to describe resources we defined in section 8.2.

8.3.1 Creating property definitions with scalars

This section explains how we define the type of a property using GraphQL's type system. We distinguish between scalar types and object types. **Scalar types** are types such as Booleans or integers. The syntax for defining a property's type is very similar to how we use type hints in Python: we include the name of the property followed by a colon, and the property's type to the right of the colon. For example, in section 8.2 we discussed that cakes have two distinct properties: `hasFilling` and `hasNutsToppingOption`, both of which are Booleans. Using GraphQL's type system, we describe these properties like this:

```
hasFilling: Boolean
hasNutsToppingOption: Boolean
```

GraphQL supports the following types of scalars:

- Strings (String): for text-based object properties.
- Integers (Int): for numerical object properties.
- Floats (Float): for numerical object properties with decimal precision.
- Booleans (Boolean): for binary properties of an object.
- Unique Identifiers (ID): for describing an object ID. Technically, ID's are strings, but GraphQL checks and ensures that the ID of each object is unique.

In addition to defining the type of a property, we can also indicate whether the property is non-nullable. Nullable properties are properties that can be set to `null` when we don't know their value. We mark a property as non-nullable by placing an exclamation mark at the end of the property definition:

```
name: String!
```

The above line defines a property `name` of type String, and it marks it as non-nullable by using an exclamation mark. This means that, whenever we serve this property from the API, it will always be a string.

Now that we've learned about properties and scalar, let's see how we use this knowledge to actually model resources!

8.3.2 Modeling resources with object types

This section explains how we use GraphQL's type system to model resources. Resources are the entities managed by the API, such as the ingredients, cakes, and beverages that we discussed in section 8.2. In GraphQL, each of these resources is modelled as an object type. **Object types** are collections of properties, and as the name indicates, we use them to define objects. To define an object type, we use the `type` keyword followed by the object name, and the list of object properties wrapped between curly braces. Listing 8.1 illustrates how we describe the cake resource as an object type. The listing contains the basic properties of the cake type, such as the ID, the name, and its price.

Listing 8.1 Definition of the Cake object type

```
type Cake {    #A
  id: ID!    #B
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!
  hasNutsToppingOption: Boolean!
}
```

#A An object type is defined using the type keyword, followed by the object name, and its properties wrapped in curly braces

#B A property is defined by declaring the property name followed by a colon, and its type on the right side of the colon. In GraphQL, ID is a type with a unique value. An exclamation mark at the end of a property indicates that the property is non-nullable

TYPES AND OBJECT TYPES. For convenience, throughout the book, we'll use the concepts of type and object type interchangeably unless otherwise stated.

Some of the property definitions in listing 8.1 end with an exclamation mark. In GraphQL, an exclamation mark means that a property is non-nullable. It means that every cake object returned by our API will contain an ID, a name, its availability, as well as the `hasFilling` and the `hasNutsToppingOption` properties. It also guarantees that none of these properties will be set to `null`. For API client developers, this information is very valuable, because they know they can count on these properties to always be present, and build their applications with that assumption. Listing 8.2 shows the definitions for the `Beverage` and the `Ingredient` types. It also shows the definition for the `Supplier` type. The `Supplier` type contains information about the business that supplies a certain ingredient, and in section 8.5.1 we'll see how we connect it with the `Ingredient` type.

Listing 8.2 Definitions of the Beverage and the Ingredient object types

```
type Beverage {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasCreamOnTopOption: Boolean!
  hasServeOnIceOption: Boolean!
}

type Ingredient {
  id: ID!
  name: String!
}

type Supplier {
  id: ID!
  mame: String!
  address: String!
  contactNumber: String!
  email: String!
}
```

Now that we know how to define object types, let's complete our exploration of GraphQL's type system by learning how to create our own custom types!

8.3.3 Creating custom scalars

This section explains how we create custom scalar definitions. In section 8.3.1 we introduced GraphQL's built-in scalars: String, Integer, Float, Boolean, and ID. In many cases, this list of scalar types is sufficient to model our API resources. In some cases, however, GraphQL's built-in scalar types might prove limited. In such cases, we can define our own custom scalar types. For example, we may want to be able to represent a date type, or a URL type, or an email address type.

Since the products API is used to manage products and ingredients and make changes to them, it'd be useful to add a `lastUpdated` property that tells us when the last time was that

a record changed. `lastUpdated` should be a `Datetime` scalar. GraphQL doesn't have a built-in scalar of that type, so we have to create our own. To declare a custom date-time scalar, we use the following statement:

```
scalar Datetime
```

We also need to define how this scalar type is validated and serialized. We define the rules for validation and serialization of a custom scalar in the server implementation, which we will be the topic of chapter 10. Listing 8.3 shows how we declare a custom `Datetime` scalar, and how we use it to define the `lastUpdated` property of the `Cake` object type.

Listing 8.3 Using a custom Datetime scalar type

```
scalar Datetime    #A

type Cake {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!
  hasNutsToppingOption: Boolean!
  lastUpdated: Datetime!    #B
}
```

#A We declare a custom Datetime scalar
#B We use our custom Datetime scalar to define the lastUpdated property of the Cake object type. We mark lastUpdated as a required property by adding an exclamation mark to its definition

This concludes our exploration of GraphQL scalars and object types. You're now in a position where you can define basic object types in GraphQL and create your own custom scalars. In the following sections, we'll learn to create connections between different object types, and we'll learn how to use lists, interfaces, enumerations, and more!

8.4 Representing collections of items with lists

This section introduces GraphQL lists. Lists are arrays of types, and they're defined by surrounding a type between square brackets. Lists are useful when we need to define properties that represent collections of items. As discussed in section 8.2, the `Ingredient` type contains a property called `description`, which contains collections of notes about the ingredient. Listing 8.4 adds the `description` property to the Ingredient object type.

Listing 8.4 Representing a list of strings

```
type Ingredient {
  id: ID!
  name: String!
  description: [String!]    #A
}
```

#A Lists are defined by surrounding a type between square brackets. This example represents a list of strings.

Look closely at the use of exclamation marks in the `description` property: we're defining it as a nullable property with non-nullable items. What does this mean? It means that, when we return an ingredient from the API, it may or may not contain a `description` field, and if the `description` field is present, it'll contain a list of strings.

When it comes to lists, you must pay careful attention to the use of exclamation marks. In list properties, we can use two exclamation marks: one for the list itself, and another one for the item within the list. To make both the list and its contents non-nullable, we use exclamation marks for both. The use of exclamation marks for list types is one of the most common sources of confusion among GraphQL users, so I've summarized in table 8.1 the possible return values for each combination of exclamation marks in a list property definition.

> **USE EXCLAMATION MARKS AND LISTS CAREFULLY!** In GraphQL, an exclamation mark indicates that a property is non-nullable. Being non-nullable means that the property needs to be present in an object and its value cannot be `null`. When it comes to lists, we can use two exclamation marks: one for the list itself, and another one for the item within the list. Different combinations of the exclamation marks will yield different representations of the property. Check out Table 8.1 to see which representations are valid for each combination of exclamation marks.

Table 8.1 Valid return values for list properties

	[Word]	[Word!]	[Word]!	[Word!]!
`null`	valid	valid	invalid	invalid
`[]`	valid	valid	valid	valid
`["word"]`	valid	valid	valid	valid
`[null]`	valid	invalid	valid	invalid
`["word", null]`	valid	invalid	valid	invalid

Now that we've learned about GraphQL's type system and list properties, we're ready to explore one of the most powerful and exciting features of GraphQL: connections between types. Move on to the next section to learn more about this topic!

8.5 Think graphs: building meaningful connections between object types

This section explains how we create connections between objects in GraphQL. One of the great benefits of GraphQL is being able to connect objects. By connecting objects, we make it clear for our API consumers how our entities are related. As we'll see in the next chapter, this makes our GraphQL API easier to be consumed.

8.5.1 Connecting types through edge properties

This section explains how we connect types by using edge properties. **Edge properties** are properties that point to another type. Types can be connected by creating a property that points to another type. As you can see in figure 8.7, a property that connects with another

object is called an *edge*. Listing 8.5 shows how we connect the `Ingredient` type with the `Supplier` type, by adding a property called `supplier` to `Ingredient` that points to `Supplier`.

Listing 8.5 Edge for one-to-one connection

```
type Ingredient {
  id: ID!
  name: String!
  supplier: Supplier!    #A
  description: String
}
```

#A We connect the Ingredient type with the Supplier type by creating a field called supplier which points to Supplier. Ingredient's supplier property an edge since it connects two types.

This is an example of one-to-one connection: a property in an object points to exactly one object. The property in this case is called an edge, because it connects the `Ingredient` type with the `Supplier` type. It's also an example of a directed connection: as you can see in figure 8.3, we can reach the `Supplier` type from the `Ingredient` type, but not the other way around, so the connection only works in one direction.

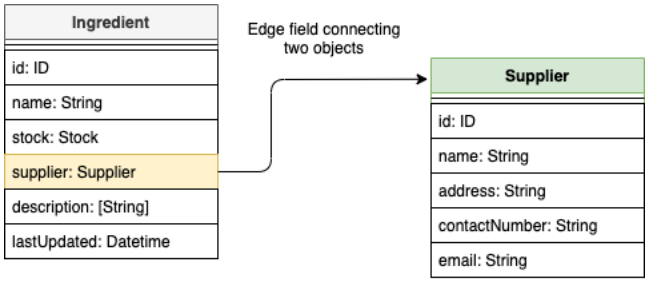


Figure 8.7 To connect the Ingredient type with the Supplier type, we add a property to Ingredient called Supplier, which points to the Supplier type. Since the Ingredient's supplier property is creating a connection between two types, we call it an edge.

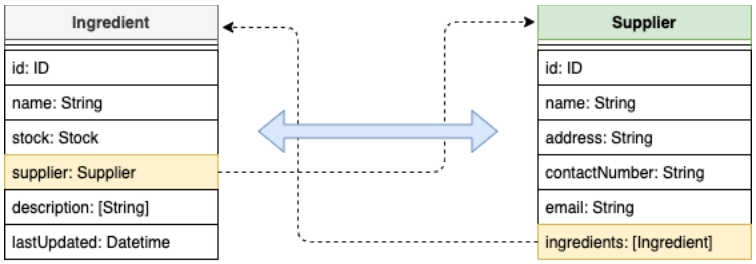


Figure 8.8 To create a bidirectional relationship between two types, we add properties to each of them that point to each other. In this example, the Ingredient's supplier property points to the Supplier type, while the Supplier's ingredients property points to a list of Ingredients.

To make the connection between the Supplier and the Ingredient bidirectional[4], we need to add a property to the `Supplier` type that points to the `Ingredient` type, as shown in listing 8.6. Since a supplier can provide more than one ingredient, the `ingredients` property points to a list of `Ingredient` types. This is an example of one-to-many connection. Figure 8.4 shows what the new relationship between the Ingredient and the Supplier types looks like.

Listing 8.6 Bidirectional relationship between Supplier and Ingredient

```
type Supplier {
  id: ID!
  name: String!
  address: String!
  contactNumber: String!
  email: String!
  ingredients: [Ingredient!]!    #A
}
```

#A To create a bidirectional relationship between the Ingredient and the Supplier types, we add a property called ingredients to Supplier, which points to a list of Ingredients.

Now that we know how to create simple connections through edge properties, let's see how we create more complex connections using dedicated types.

8.5.2 Creating connections with through types

This section discusses through types. **Through types** are are types that tell us how other object types are connected: they add additional information about the connection itself. We'll use through types t oconnect our products, cakes and beverages, with their ingredients. We could connect them by adding a simple list of ingredients to Cake and Beverage, as shown in figure 8.9, but this wouldn't tell us how much of each ingredient goes into the making of each product.

[4] In the literature about GraphQL, you'll often find a digression about how GraphQL is inspired by graph theory, and how we can use some of the concepts from graph theory to illustrate the relationships between types. Following that tradition, the bidirectional relationship we refer to here would be an example of an undirected graph, since the `Supplier` type can be reached from the `Ingredient` type, and vice versa. For a good discussion of graph theory in the context of GraphQL, see Eve Porcello and Alex Banks, *Learning GraphQL, Declarative Data Fetching for Modern Web Apps*, (O'Reilly, 2018), pp. 15-30.

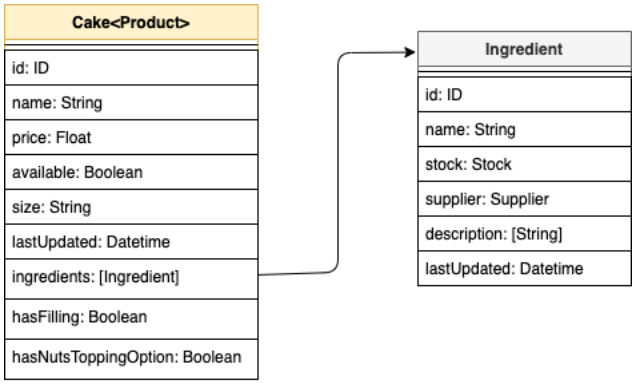


Figure 8.9 We can express Cake's ingredients field as a list of Ingredient types, but that wouldn't tell us how much of each ingredient goes into the making of a specific cake.

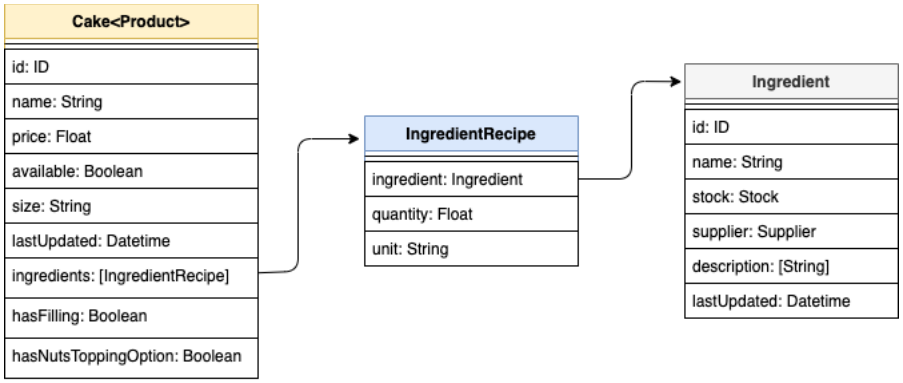


Figure 8.10 To express how an Ingredient is connected with a Cake, we use the IngredientRecipe through type, which allows us to detail how much of each ingredient goes into the making of a cake.

To connect cakes and beverages with their ingredients, we'll use a through type called `IngredientRecipe`. As you can see in figure 8.10, `IngredientRecipe` has three properties: the ingredient itself, as well as its amount, and the unit in which the amount is measured. This gives us more meaningful information about how our products relate to their ingredients. Listing 8.7 shows how we define the `IngredientRecipe` through type and use it in the `Cake` and `Beverage` types.

Listing 8.7 Through types represent a relationship between two types

```
type IngredientRecipe {    #A
  ingredient: Ingredient!
  quantity: Float!
  unit: String!
}

type Cake {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!
  hasNutsToppingOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe!]!    #B
}

type Beverage {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasCreamOnTopOption: Boolean!
  hasServeOnIceOption: Boolean!
  lastUpdated: Datetime!    #B
  ingredients: [IngredientRecipe!]!
}
```

#A We declare the IngredientRecipe through type. A through type is just a normal object type, so we declare it with the type keyword.

#B We use the IngredientRecipe through type in the ingredients property of both Cake and Beverage.

By creating connections between different object types, we give our API consumers the ability to explore our data by just following the connecting edges in the types. And by creating bidirectional relationships, we give users the ability to traverse our data graph back and forth. This is one of the most powerful features of GraphQL, and it's worth always spending the time to design meaningful connections across our data.

Talking about data and connections, more often than not, we'll need to create properties that return multiple types. For example, we could have a property that returns either cakes and beverages. This is the topic of the next section. Move on to discover how we deal with these situations!

8.6 Combining different types through unions and interfaces

This section discusses how we cope with situations where we have multiple types of the same entity. You'll often have to deal with properties that point to a collection of multiple types. What does this mean in practice, and how does it work? Let's look at an example from the products API!

In the products API, `Cake` and `Beverage` are two types of products. In section 8.4.2, we saw how we connect `Cake` and `Beverage` with the `Ingredient` type. But how do we connect

`Ingredient` to `Cake` and `Beverage`? We could simply add a property called `products` to the `Ingredient` type, which points to a list of `Cakes` and `Beverages`, like this:

```
products: [Cake, Beverage]
```

This works, but it doesn't allow us to represent Cakes and Beverages as a single product entity. Why would we want to do that? Because of the following reasons:

1. Cake and Beverage are the same thing: a product, and as such, it makes sense to treat them as the same entity.
2. As we'll see in sections 8.8 and 8.9, we'll have to refer to our products in other parts of the code, and it'll be very helpful to be able to use one single type for that.
3. If we add new types of products to the system in the future, we don't want to have to change all parts of the specification that refer to products. Instead, we want to have a single type that represents them all, and update only that type.

GraphQL offers two ways to bring various types together under a single type: unions and interfaces. Let's look at each of them in detail.

Interfaces are useful when we hacve types that share properties in common. This is the case of the Cake and the Beverage types, which share most of their properties. GraphQL interfaces are similar to class interfaces in programming language, such as Python: they define a collection of properties which must be implemented by other types. Listing 8.8 shows how we use an interface to represent the collection of properties shared by the `Cake` and `Beverage` . As you can see, the `Cake` and `Beverage` types must implement all the properties defined in the `ProductInterface` type. By looking at the `ProductInterface` type, any user of our API can quickly get an idea of which properties will be accessible on both the `Beverage` and the `Cake` types.

Listing 8.8 Representing common properties through interfaces

```
interface ProdcutInterface {    #A
  id: ID!
  name: String!
  prices: [Prices!]!
  ingredients: [IngredientRecipe!]
  available: Boolean!
}

type Cake implements ProdcutInterface {    #B
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!    #C
  hasNutsToppingOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe!]!
}

type Beverage implements ProdcutInterface {    #D
  id: ID!
  name: String!
```

```
  price: Float
  available: Boolean!
  hasCreamOnTopOption: Boolean!
  hasServeOnIceOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe!]!
}
```

#A We declare the ProductInterface interface type by using the interface keyword.
#B The Cake type implements the ProductInterface interface, which means we need to define all the properties that belong in the ProductInterface type.
#C In addition to implementing ProductInterface's properties, we can add additional properties specific to Cake, just as hasFilling.
#D The Beverage type also implements the ProductInterface interface.

By creating interfaces, we make it easier for our API consumers to understand what the common properties shared by our product types are. As we'll see in the next chapter, interfaces also make the API easier to consume.

While interfaces help us define the common properties of various types, unions help us to bring various types under the same type. This is very helpful when we want to treat various types as a single entity. In the products API, we want to be able to treat the Cake and Beverage types as a single Product type, and unions allow us to do that. A union type is the combination of different types using the pipe (|) operator. Listing 8.9 shows how we combine `Beverage` and `Cake` into a `Product` type using a union.

Listing 8.9 Union of different types

```
type Cake implements ProdcutInterface {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!
  hasNutsToppingOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe!]!
}

type Beverage implements ProdcutInterface {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasCreamOnTopOption: Boolean!
  hasServeOnIceOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe!]!
}

union Product = Beverage | Cake    #A
```

#A We create a union type by using the union keyword, and by joining any number of types using the pipe (|) operator.

Using unions and interfaces makes our API easier to maintain and to consume. If we ever add a new type of product to the API, we can make sure it offers a similar interface to `Cake` and `Beverage` by making it implements the `ProductInterface` type. And by adding the new product to the `Product` union, we make sure it's available on all operations that use the Product union type.

Now that we know how to combine multiple object types, it's time to learn how we constrain the values of object type properties through enumerations!

8.7 Constraining property values with enumerations

This section covers GraphQL's enumeration type. Technically, an enumeration is a specific type of scalar which can only take on a predefined number of values. Enumerations are useful in properties which can only accept a value from a constrained list of choices.

In the products API, we need enumerations for expressing the amounts of the ingredients. For example, in section 8.5.2, we defined a through type called `IngredientRecipe`, which indicates the amount of each ingredient that goes into a product. `IngredientRecipe` expresses amounts in terms of quantity per unit of measure. We can measure ingredients in different ways. For example, we can measure milk in pints, liters, ounces, gallons, and others. For the sake of consistency, we want to ensure that everybody uses the same units to describe the amounts of our ingredients, so we'll create an enumeration type called `MeasureUnit` that can be used to constrain the values for the unit property. Listing 8.10 shows how we define the `MeasureUnit` enumeration type and use it in the `IngredientRecipe`'s unit property.

Listing 8.10 Using enumeration types

```
enum MeasureUnit {    #A
  LITERS    #B
  KILOGRAMS
  UNITS
}

type IngredientRecipe {
    ingredient: Ingredient!
    quantity: Float!
    unit: MeasureUnit!    #C
}
```

#A We declare an enumeration by using the enum keyword.
#B The properties of an enumeration are the list of possible values the enumeration can take.
#C We define the unit property as an enumeration by pointing it to an enumeration type.

We also want to use the `MeasureUnit` enumeration to describe the available stock of an ingredient. To do so, we define a Stock type, and we use it to define the `stock` property of the `Ingredient` type, as shown in listing 8.11.

Listing 8.11 Using enumeration types

```
type Stock {    #A
  amount: Float!
  unit: MeasureUnit!    #B
}

type Ingredient {
  id: ID!
  name: String!
  stock: Stock    #C
  products: [Product!]!
  supplier: Supplier!
  description: String
}
```

#A We declare the Stock type to help us express information about the available stock of an ingredient.
#B Stock's unit property is an enumeration.
#C We connect the Ingredient type with the Stock type through Ingredient's stock property.

Enumerations are very useful to ensure that certain values remain consistent through the interface. Enumerations are useful because they enforce the constraints on the values that our properties can take. This helps to avoid errors that happen when you let users choose and write those values by themselves.

This concludes our journey through GraphQL's type system! Types are the building blocks of an API specification, but without a mechanism to query them or interact with them, our API is very limited. To use our types, we need to explore two new type GraphQL types: queries and mutations. Those will be the topic of the rest of the chapter!

8.8 Defining queries to serve data from the API

This section introduces GraphQL queries. **Queries** are operations that allow us to fetch or read data from the server. Serving data is one of the most important functions of any web API, and GraphQL offers great flexibility to create a powerful query interface. Queries correspond to the group of read operations that we dsicussed in section 8.2. As a reminder, these are the query operations that the products API needs to support:

- `allProducts()`
- `allIngredients()`
- `products()`
- `product()`
- `ingredient()`

We'll work first on the `allProducts` query, since it's the simplest, and then move on to the `products` query. As we work on `products`, we'll see how we add arguments to our query definitions, we'll learn about pagination, and finally, we'll learn how to refactor our query parameters into their own type to improve readability and maintenance.

The specification of a GraphQL query looks very similar to the signature definition of a Python function: we define the query name, optionally define a list of parameters for the query between parentheses and specify the return type after a colon. Listing 8.12 shows the simplest query in the products API: the `allProducts` query, which returns a list of all

products. `addProducts` doesn't take any parameters and simply returns a list of all products that exist in the server.

Listing 8.12 Simple GraphQL query to return a list of products

```
type Query {    #A
  allProducts: [Products!]!    #B
}
```

#A All queries are defined under the Query object type.
#B We define the allProducts query. After the colon, we indicate what the return type of the query is.

`allProducts` returns a list of all products that exist in the CofeeMesh database. Such a query is useful if we want to run an exhaustive analysis of all products, but in real life, our API consumers want to be able to filter the results. They can do that by using the products query, which according to the requirements we gathered in section 8.2, returns a filtered list of products.

Query arguments are defined within parentheses, similar to how we define the parameters of a Python function. Listing 8.13 shows how we define the `products` query. It includes arguments that allows our API consumers to filter products by availability, or by maximum and minimum price. All the arguments are optional. API consumers are free to use any or all of the query arguments, or none. If they don't specify any of the arguments when using the `products` query, they'll get a list of all the products.

Listing 8.13 Simple GraphQL query to return a list of products

```
type Query {
  products(available: Boolean, maxPrice: Float, minPrice: Float): [Product!]    #A
}
```

#A Query parameters are defined within parenthesis.

In addition to filtering the list of products, API consumers are likely to want to be able to sort the list and paginate the results. Pagination is the ability to deliver the result of a query in different sets of a specified size, and it's commonly used in APIs to ensure that API clients receive a sensible amount of data in each request. As illustrated in figure 8.11, if the result of a query yields 10 or more records, we can divide the query result into groups of five items each, and serve one set at a time. Each set is called a *page*.

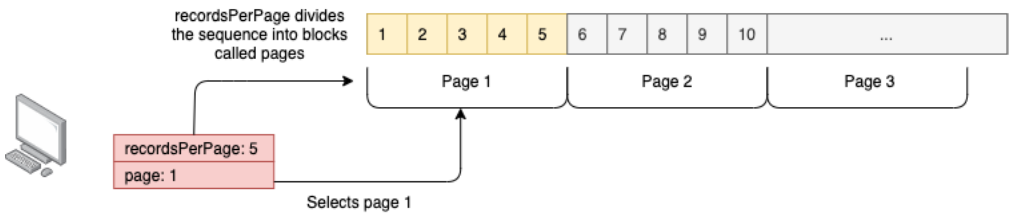


Figure 8.11 A more common approach to pagination is to let users decide how many results per page they want to see, and let them select the specific page they §want to get.

We enable pagination by adding a `resultsPerPage` argument to the query, as well as a `page` argument. To sort the result set, we expose a `sort` argument. The below snippet shows in bold characters the changes to the products after we add these arguments:

```
products(available: Boolean, maxPrice: Float, minPrice: Float, sort: SortingOrder,
       resultsPerPage: Int, page: Int): [Product!]
```

Offering numerous query arguments gives a lot of flexibility to our PAI consumers, but it can be cumbersome to set values for all of them. We can our API easier to use by setting default values for some of the arguments. We'll set a default sorting order so that the results always appear sorted by default, as well as a default value for the `restultsPerPage` argument, and a default value for the `page` argument. Listing 8.14 shows how we assign default values to some of the arguments in the products query. The listing also includes a `SortingOrder` enumeration which constrains the values of the `sort` argument to either `ASCENDING` or `DESCENDING`.

Listing 8.14 Setting default values for query arguments

```
enum SortingOrder {    #A
  ASCENDING
  DESCENDING
}

type Query {
  products(
    maxPrice: Float, minPrice: Float, available: Boolean=true,    #B
    sort: SortingOrder=DESCENDING, resultsPerPage: Int=10,    #C
    page: Int=1
  ): [Product!]
}
```

#A We declare the SortingOrder enumeration to constrain the values a user can choose for sorting.
#B To make our queries more user-friendly, we set default values for some of the parameters.
#C We constrain the values of the sort parameter by defining it as an enumeration type pointing to the SortingOrder enumeration.

The signature of the `products` query is becoming a bit cluttered. If we keep on adding arguments to it, it'll become difficult to read and maintain. To improve readability, we can refactor the arguments out of the query specification into their own type. In GraphQL, we can define lists of parameters by using input types. Input types have the same look and feel as any other GraphQL object type, but they're meant for use as input for queries and mutations. Listing 8.15 shows how we refactor `products`'s arguments into a `ProductsFilter` input type.

Listing 8.15 Refactoring query arguments into input types

```
input ProductsFilter {    #A
  maxPrice: Float    #B
  minPrice: Float
  available: Boolean=true,    #C
  sort: SortingOrder=DESCENDING
  resultsPerPage: Int=10
  page: Int = 1
```

```
}

type Query {
  products(input: ProductsFilter): [Product!]    #D
}
```

#A We declare an input type by using the input keyword.
#B We define the properties of the input type just like we do for any other object type.
#C The properties of input types can have default values too.
#D We set the input parameter of the products query to the ProductsFilter input type.

The remaining API queries, namely allIngredients, product, and ingredient, are shown in listing 8.16 in bold characters. allIngredients returns a full list of ingredients and therefore takes no arguments, as in the case of the allProducts query. Finally, product and ingredient return a single product or ingredient by ID, and therefore have a required id argument of type ID. If a product or ingredient is found for the provided ID, the queries will return the details of the requested item, otherwise they'll return null.

Listing 8.16 Specification for all the queries in the products API

```
type Query {
  allProducts: [Product!]!
  allIngredients: [Ingredient!]!
  products(input: ProductsFilter!): [Product!]!
  product(id: ID!): Product    #A
  ingredient(id: ID!): Ingredient
}
```

#A The product query returns a Product type if a product can be found for the provided id, otherwise it returns null. For this reason, the Product return type is nullable.

Now that we know how to define queries, it's time to learn about mutations, which is the topic of the next section!

8.9 Altering the state of the server with mutations

This section introduces GraphQL mutations. **Mutations** are operations that allow us to trigger actions that change the state of the server. While the purpose of a query is to let us fetch data from the server, mutations allow us to create new resources, to delete them, or to alter their state. Mutations have a return value, which can be scalar, such as a Boolean, or an object. This allows our API consumers to verify that the operation completed successfully, and also fetch any values generated by the server, such as records IDs.

In section 8.2, we discussed that the products API needs to support the following operations for adding, deleting, and updating resources in the server:

- `addIngredient()`
- `updateStock()`
- `addProduct()`
- `updateProduct()`
- `deleteProduct()`

In this section, we'll document the `addProduct`, the `updateProduct`, and the `deleteProduct` mutations. The specification for the other mutations is similar to these ones, and you can check them out in the GitHub repository provided with this book.

A GraphQL mutation looks similar to the signature of a function in Python: we define the name of the mutation, describe its parameters between parenthesis, and provide its return type after a colon. Listing 8.17 shows the specification for the `addProduct` mutation. `addProduct` accepts a long list of arguments, and it returns a `Product` type. All the arguments are optional except `name` and `type`. We use `type` to indicate what kind of product we're creating, whether a cake or a beverage. We also include a `ProductType` enumeration to constrain the values of the `type` argument to either `cake` or `beverage`. Since this mutation is used to create cakes and beverages, we allow using the specific properties of each type, namely `hasFilling` and `hasNutsToppingOption` for cakes, as well as `hasCreamOnTopOption` and `hasServeOnIceOption` for beverages, but we set them by default to `false` to make the mutation easier to use.

Listing 8.17 Defining a GraphQL mutation

```
enum ProductType {    #A
  cake
  beverage
}

type Mutation {    #B
  addProduct(name: String!, price: String, size: String, ingredients: [ID!],    #C
    hasFilling: Boolean = false, hasNutsToppingOption: Boolean = false,
    hasCreamOnTopOption: Boolean = false, hasServeOnIceOption: Boolean = false
  ): Product!    #C
}
```

#A We declare a ProductType enumeration to constrain the values a user can give for addProduct's type parameter.
#B Mutations are declared under the Mutation object type.
#C We declare the addProduct mutation.
#D We specify the return type of the addProduct mutation, which is a non-nullable Product type.

You'd agree that the signature definition of the `addProduct` mutation looks a bit cluttered. We can improve readability and maintainability by refactoring the list of parameters into their own type. Listing 8.18 shows how we refactor the `addProduct` mutation by moving the list of parameters into an input type. `AddProductInput` contains all the optional parameters that can be set when we create a new product. We set aside the `name` parameter, which is the only required parameter when we create a new product. As we'll see shortly, this allows us to reuse the `AddProductInput` input type in other mutations that don't require the `name` parameter.

Listing 8.18 Refactoring parameters with input types

```
input AddProductInput {    #A
  price: String    #B
  size: String
  ingredients: [ID!]
  hasFilling: Boolean = false    #C
  hasNutsToppingOption: Boolean = false
  hasCreamOnTopOption: Boolean = false
  hasServeOnIceOption: Boolean = false
}


type Mutation {
  addProduct(name: String!, input: AddProductInput): Product!    #D
}
```

#A We declare an input type by using the input keyword.
#B We define the properties of an input type just as we define the properties of any other object type.
#C The properties of input types can have default values too.
#D We point addProduct's input parameter to the AddProduct input type.

Input types not only help us make our specification more readable and maintainable, but they also they also allow us to create reusable types. We can reuse the `AddProductInput` input type in the signature of the `updateProduct` mutation. When we update the configuration for a product, we may want to change only some of its parameters, such as the name, the price, or its ingredients. The snippet below shows how we reuse the `AddProductInput` parameters in `updateProduct`. In addition to `AddProductInput`, we also include a mandatory product `id` parameter, which is necessary identify the product we want to update. We also include the `name` parameter, which in this case is optional:

```
type Mutation {
  updateProduct(id: ID!, name: String, input: AddProductInput): Product!
}
```

Let's now look at the `deleteProduct` mutation. `deleteProduct` removes a product from the catalogue. To do that, the user must provide for the product they intend to delete. If the operation is successful, the mutation returns `true`, and otherwise it returns `false`. The below snippet shows the specification for the `deleteProduct` mutation:

```
deleteProduct(id: ID!): Boolean!
```

This concludes our journey through GraphQL's Schema Definition Language! You're now equipped with everything you need to define your own API Schemas. In chapter 9, we'll learn how to launch a mock server using the products API specification, and how to consume and interact with the GraphQL API.

8.10 Summary

- GraphQL is a popular protocol for building web APIs. GraphQL shines in scenarios where it's important to give API clients full control over the data they want to fetch, and in situations where we have highly interconnected data.
- A GraphQL API specification is called a schema, and it's written using the Schema Definition Language (SDL).
- We use GraphQL's scalar types to define the properties of an object type. GraphQL's built-in scalar types are Booleans, Strings, Floats, Integers, and IDs. In addition to this, we can also create our own custom scalar types.
- GraphQL's object types are collections of properties, and they typically represent the resource or entities managed by the API server.
- We can connect objects by using edge properties, namely properties that point to another object, and by using through types. Through types are object types that add additional information about how two objects are connected.
- To constrain the values of a property, we use enumeration types.
- GraphQL queries are operations that allow API clients to fetch data from the server.
- GraphQL mutations are operations that allow API clients to trigger actions that change the state of the server.
- When queries and mutations have long lists of parameters, we can refactor them into input types to increase readability and maintainability. Input types can also be reused in more than one query or mutation.

9

Consuming GraphQL APIs

This chapter covers

- **Running a GraphQL mock server to test our API design**
- **How to use the GraphiQL client to explore and consume a GraphQL API**
- **Running queries and mutations against a GraphQL API**
- **Consuming a GraphQL API programmatically using cURL and Python**

This chapter teaches you how to consume GraphQL APIs. As we learned in chapter 8, GraphQL offers a query language for web APIs, and in this chapter, you'll learn how to use this language to run queries on the server. In particular, you'll learn how to make queries against a GraphQL API. You'll learn to explore a GraphQL API so that we can discover its available types, queries, and mutations. Understanding how GraphQL APIs work from the client side is a very important step towards mastering GraphQL.

Learning to interact with GraphQL APIs will help you learn to consume the APIs exposed by other vendors, it'll let you run tests against your own APIs, and it'll help you design better APIs. You'll learn to use the GraphiQL client to explore and visualize a GraphQL API. As you'll see, GraphiQL offers an interactive query panel that makes it easier to run queries on the server.

To illustrate the concepts and ideas behind GraphQL's query language, we'll run practical examples using the products API that we designed in chapter 8. Since we only have the API specification for the products API at this moment, we'll learn to run a mock server. As you'll see, running mock servers is an important part of the API development process, as it makes testing and validating an API design so much easier. Finally, you'll also learn to run queries against a GraphQL API programmatically using tools such as cURL and Python.

9.1 Running a GraphQL mock server

In this section, we explain how we can run a GraphQL mock server to explore and test our API. A **mock server** is fake server that emulates the behavior of the real server, offering the same endpoints and capabilities, but using fake data. For example, a mock server for the products API is a server that mimics the implementation of the products API, offering the same exact same interface that we developed in chapter 8.

> **DEFINITION** **Mock servers** are fake servers that mimic the behavior of a real server. Mock servers are commonly used for developing API clients while the backend is being implemented. You can launch a mock server using the specification for an API. Mock servers use fake data and typically don't persist data.

Mock servers are instrumental in the development of web APIs, since they allow our API consumers to start working on the client-side code while we work on the backend implementation. In this section, we'll run a mock server on the products API. The only thing we need to run a mock server is the API specification, which we developed in chapter 8. You'll find the API specification under ch08/schema.graphql in the GitHub repository for this book.

You choose from among many different libraries to run a GraphQL mock server. In this chapter, we'll use graphql-faker (https://github.com/APIs-guru/graphql-faker), which is one of the most popular GraphQL mocking tools. To install graphql-faker, run the following command:

```
$ npm install graphql-faker
```

This will create a `package-lock.json` file under your current directory, as well as a `node_modules` folder. `package-lock.json` contains information about the dependencies installed together with graphql-faker, while `node_modules` is the directory where those dependencies are installed. To run the mock server, execute the following command:

```
$ ./node_modules/.bin/graphql-faker schema.graphql
```

Graphql-faker normally runs on port 9002 and it exposes three endpoints:

- /editor: an interactive editor where you can develop your GraphQL API.
- /graphql: a GraphiQL interface to your GraphQL API. This is the interface that we'll use to explore the API and run our queries.
- /voyager: an interactive display of your API which helps you understand the relationships and dependencies between your types (see figure 9.1 for an illustration the products API).

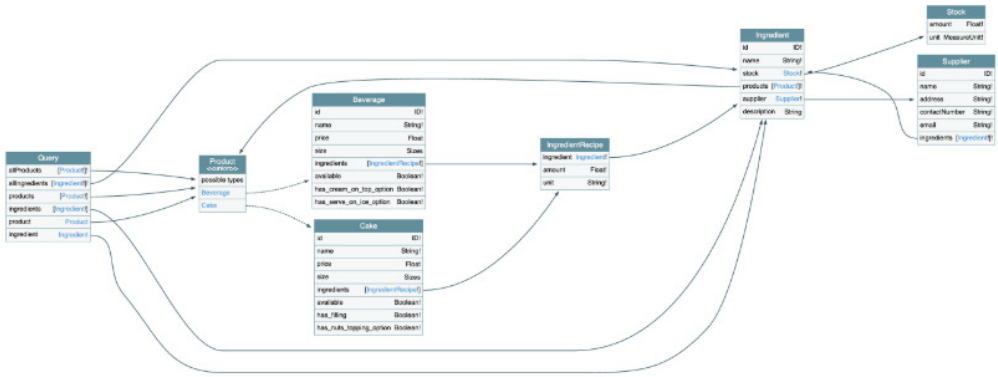


Figure 9.1 Voyager UI for the products API. This UI shows the relationships between object types captured by the queries available in the API. By following the connecting arrows, you can see which objects we can reach from each query.

To start exploring and testing the products API, visit the following address in your browser: http://localhost:9002/graphql (if you're running graphql-faker in a different port, your URL will look different). This endpoint loads a GraphiQL interface for our products API. Figure 9.2 illustrates what this interface looks like and highlights the most important elements in it.

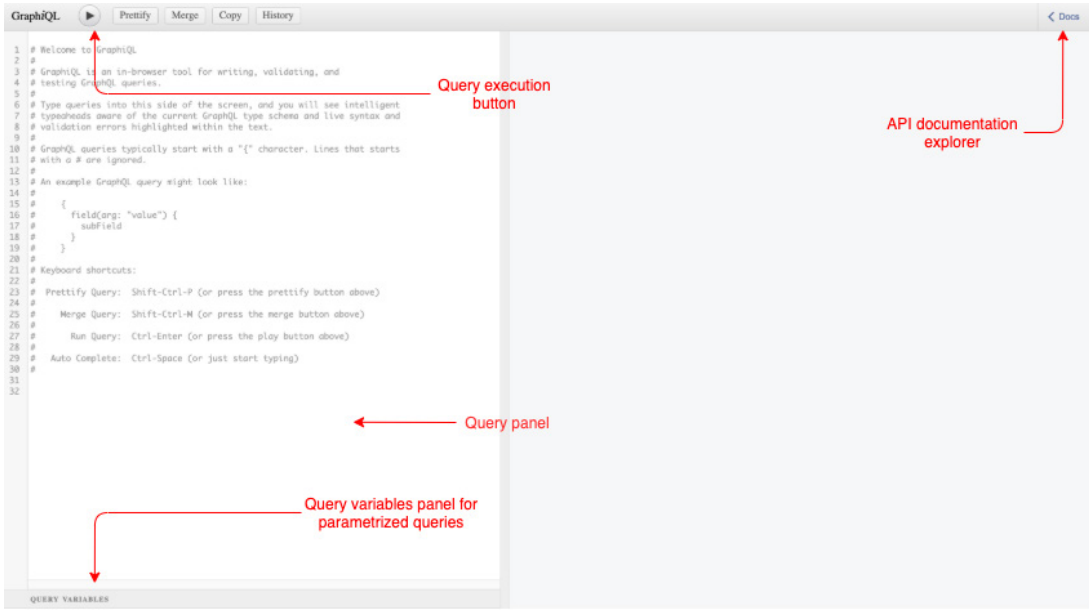


Figure 9.2 API documentation explorer and query panel interface in GraphiQL.

To discover the queries and mutations exposed by the API, click on the Docs button on the top-right corner of the UI. Upon clicking the Docs button, a side navigation bar will popup offering two choices: queries or mutations (see figure 9.3 for an illustration). If you select queries, you'll see the list of queries exposed by the server, together with their return types. You can further click on the return types to explore their properties, as you can see in figure 9.3. In the next section, we'll start testing the GraphQL API!

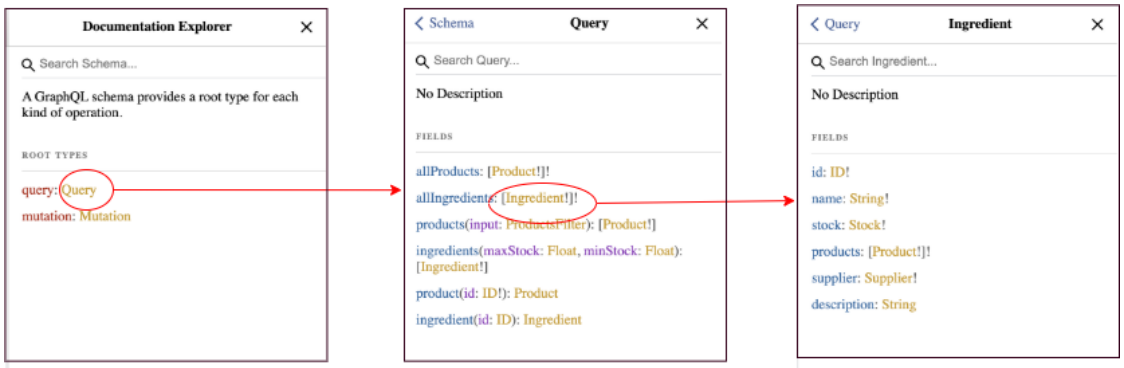


Figure 9.3 Clicking through the Documentation Explorer in GraphiQL, you can inspect all the queries and mutations available in the API, as well as types they return and their properties.

9.2 Introducing GraphQL queries

In this section, we learn to consume a GraphQL API by running queries using GraphiQL. We'll start with simple queries that don't require any parameters, and then we'll move on to queries with parameters.

9.2.1 Running simple queries

In this section, we introduce simple queries that don't take any parameters. The products API offers two queries of this type: `allProducts`, which returns a list of all products offered by CoffeeMesh, and `allIngredients`, which returns a list of all the ingredients.

We'll use GraphiQL to run queries against the API. To run the query, head over to the query editor pane in the GraphiQL UI, which is illustrated in Figure 9.2. Listing 9.1 shows how we run the `allIngredients` query. As you can see, to run a query, we must use the name of the query operation followed by curly braces. Within the curly braces, we declare the selection of properties that we want to get from the server. The block within curly braces is called a **selection set**. GraphQL queries must always include a selection set. If you don't include the selection set, you'll get an error response from the server. In listing 9.1, we're selecting only the name of each ingredient. The text representing the query is called **query document**.

9.1 Query document running the `allIngredients` query

```
{    #A
  allIngredients {    #B
    name    #C
  }
}
```

#A All queries must be wrapped within curly braces
#B To run a query without parameters, we include the name of the query followed by curly braces of the selection set
#C The selection set includes the properties we want to fetch for each object returned by the query

A response to a successful query from a GraphQL API contains a JSON document with a "data" field which wraps the query result, as you can see in listing 9.2. An unsuccessful query results in a JSON document which contains an "error" key. Since we're running a mock server, the API returns random values.

9.2 Example of successful response for the `allIngredients` query

```
{
  "data": {    #A
    "allIngredients": [    #B
      {
        "name": "string"
      },
      {
        "name": "string"
      }
    ]
  }
}
```

#A A successful response includes a "data" key
#B The result of the query is indexed under a key named after the query itself

Now that we know the basics of GraphQL queries, let's spice up our queries by adding parameters!

9.2.2 Running queries with parameters

This section explains how we use parameters in GraphQL queries. `allIngredients` is a simple query that doesn't take any parameters. Now let's see how we can run a query that requires a parameter. One example of such query is the `ingredient` query, which requires an `id` parameter. Listing 9.3 shows how we can call the `ingredient` query with a random ID.

9.3 Running a query with a required parameter

```
{
  ingredient(id: "asdf") {    #A
    name
  }
}
```

#A To run a query with parameters, we include the parameters between parentheses

Now that we know how to run queries with parameters, let's look at the kind of problems we can run into when running queries, and how to deal with them!

9.2.3 Understanding query errors

This section explains some of the most common errors you'll find when running GraphQL queries, and it teaches you how to read and interpret them.

If you omit the required parameter when running the ingredient query, you'll get an error from the API. Error responses include an error key pointing to a list of all the errors found by the server. Each error is an object with the following keys:

- message: includes a human-readable description of the error.
- location: specifies where in the query the error was found, including the line and column.

Listing 9.4 shows what happens when you run the query with empty parentheses. As you can see, get a syntax error with a somewhat cryptic message, saying "Expected Name, found)". This is a common error that occurs whenever you make a syntax error in GraphQL. In this case, it means that GraphQL was expecting a parameter after the opening parenthesis, but instead it found a closing parenthesis ")".

9.4 Missing query parameter errors

```
# Query:
{
  Ingredient() {    #A
    name
  }
}

# Error:
{
  "errors": [    #B
    {
      "message": "Syntax Error: Expected Name, found )",    #C
      "locations": [    #D
        {
          "line": 2,    #E
          "column": 14    #F
        }
      ]
    }
  ]
}
```

#A We run the `ingredient` query without the required parameter `id`
#B An unsuccessful response includes an "errors" key
#C We get a generic syntax error
#D The precise location of the error in our query
#E The error was found the second line of our query document
#F The error was found at the 14th character in the second line, which is represented by the closing parenthesis

On the other hand, if you run the ingredient query without any parentheses at all, as shown in listing 9.5, you'll get an error specifying that you missed the required parameter `id`.

USE OF PARENTHESES IN GRAPHQL QUERIES AND MUTATIONS. In GraphQL, the parameters of a query are defined within parentheses. If you run a query with required parameters, such as `ingredient`, you must include the parameters within parentheses (see listing 9.3). Failing to do so will throw an error (see listings 9.4 and 9.5). if you run a query without parameters, you must omit the parentheses. For example, when we run the `allIngredients` query, we omit parentheses (see listing 9.1), since `allIngredients` doesn't require any parentheses.

9.5 Missing query parameter errors

```
# Query:
{
  ingredient() {    #A
    name
  }
}

# Error:
{
  "errors": [
    {
      "message": "Field \"ingredient\" argument \"id\" of type \"ID!\" is required, but it
       was not provided.",    #B
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ]
    }
  ]
}
```

#A We run the `ingredient` query with parentheses but omitting the required parameter
#B We an error message specifying that the required `id` parameter wasn't found in the query
#C The error was found in the second line of our query document
#D The error was found at the 3rd character of the second line, which is represented by the character "l"

Now that we know how to read and interpret error messages when we make mistakes in our queries, let's explore queries that return multiple types. That's the topic of the next section!

9.3 Using fragments in queries

This section explains how we run queries that return multiple types. The queries that we've seen so far in this chapter are simple since they only return one type, which is `Ingredient`. However, our product-related queries, such as `allProducts` and `product`, return the `Product` union type, which is the combination of the `Cake` and the `Beverage` types. How do we run our queries in this case?

When a GraphQL query returns multiple types, we must create selection sets for each type. For example, if you try to run the `allProducts` query with a single selector, as shown in listing 9.6, you'll get the error message shown in the listing.

9.6 Calling allProducts with a single selector set

```
# Query
{
  allProducts {    #A
    name    #B
  }
}

# Error message
{
  "errors": [    #C
    {
      "message": "Cannot query field \"name\" on type \"Product\". Did you mean to use an
       inline fragment on \"ProductInterface\", \"Beverage\", or \"Cake\"?",    #D
      "locations": [
        {
          "line": 3,    #E
          "column": 5    #F
        }
      ]
    }
  ]
}
```

#A We run the `allProducts` query without parameters
#B We include the name property in the selection set
#C The server response is unsuccessful and therefore it includes an `error` keyword
#D The error message says that we're missing an inline fragment in the query
#E The error was found in the 3rd line of the query document
#F The error was found at the 5th position of the 3rd line, represented by the character "n"

The error message in listing 9.6 asks you whether you meant to use an inline fragment on either `ProductInterface`, `Beverage`, or `Cake`. What is an inline fragment? An **inline fragment** is an anonymous selection set on a specific type. The syntax for inline fragments includes three dots (the spread operator in JavaScript) followed by the `on` keyword and the type on which the section set applies, and a selection of properties between curly braces:

```
...on ProductInterface {
      name
    }
```

Listing 9.7 fixes the `allProducts` query by adding inline fragments that select properties on the `ProductInterface`, the `Cake`, and the Beverage types. `allProducts`'s return type is `Product`, which is the union of `Cake` and `Beverage`, so we can select properties from both types. From the specification, we also know that `Cake` and `Beverage` both implement the `ProductInterface` interface type, so we can conveniently select common properties to both `Cake` and `Beverage` directly on the interface.

9.7 Adding inline fragments for each return type

```
{
  allProducts {
    ...on ProductInterface {    #A
      name
    }
    ...on Cake {    #B
      hasFilling
    }
    ...on Beverage {    #C
      hasCreamOnTopOption
    }
  }
}
```

#A We include an inline fragment with a selection set on the `ProductInterface` type
#B We include an inline fragment with a selection set on the `Cake` type
#C We include an inline fragment with a selection set on the `Beverage` type

Listing 9.7 uses inline fragments, but the real benefit of fragments is we can define them as standalone variables. This makes fragments reusable, and it also makes our queries more readable. Listing 9.8 shows how we can refactor listing 9.7 to use standalone fragments. The queries are so much cleaner! In real life situations, you're likely to work with large selection sets, so organizing your fragments into standalone, reusable pieces of code will make your queries easier to read.

9.8 Using standalone fragments

```
{
  allProducts {
    ...commonProperties
    ...cakeProperties
    ...beverageProperties
  }
}

fragment commonProperties on ProductInterface {
  name
}

fragment cakeProperties on Cake {
  hasFilling
}

fragment beverageProperties on Beverage {
  hasCreamOnTopOption
}
```

Now that we know how to deal with queries that return multiple object types, let's take our querying skills to the next level. In the next section, we'll learn to run queries with a specific type of parameter called input parameter.

9.4 Running queries with input parameters

This section explains how we run queries with input type parameters. In section 8.8, we learned that input types are similar to object types, but they're meant for use as parameters for a GraphQL query or mutation. One example of input type in the products API is `ProductsFilter`, which allows us to filter products by factors such as availability, minimum or maximum price, and others. `ProductsFilter` is the parameter of the `products` query. How do we call the `products` query?

When a query takes parameters in the form of an input type, the query's input type parameter must be passed in the form of an input object. This may sound complicated, but it's actually very simple. To illustrate how it works, listing 9.9 shows how we call the `products` query using `ProductsFilter`'s `maxPrice` parameter. To use any of the parameters in the input type, we simply wrap them within curly braces.

9.9 Calling a query with a required parameter

```
{
  products(input: {maxPrice: 10}) {    #A
    ...on ProductInterface {    #B
      name
    }
  }
}
```

#A We run the products query passing in `ProductFilter`'s `maxPrice` parameter
#B We add an inline fragment on the `ProductInterface` type

Now that we know how to call queries with input parameters, let's take a deeper look at the relationships between the objects defined in the API specification, and see how we can build queries that allow us to traverse our data graph. That's the topic of the next section!

9.5 Navigating the API graph

This section explains how we select properties from multiple types by leveraging their their connections. In section 8.5, we learned to create connections between object types by using edge properties and through types. These connections allow API clients to traverse the graph of relationships between the resources managed by the API. For example, in the products API, the `Cake` and `Beverage` types are connected with the `Ingredient` type by means of a through type called `IngredientRecipe`. By leveraging this connection, we can run queries that fetch information about the ingredients related to each product. In this section, we'll learn to build such queries.

In our queries, whenever we add a selector for a property that points to another object type, we must include a nested selection set for said object type. For example, if we add a selector for the `ingredient` property on the `ProductInterface` type, we have to include a selection set with any of the properties in `IngredientRecipe` nested within the `ingredients` property. Listing 9.10 shows how we include a nested selection set for the `ingredients` property of `ProductInterface` in the `allProducts` query. The query selects each product's name, together with the names of the ingredients in each product recipe.

9.10 Querying nested object types

```
{
  allProducts {
    ...on ProductInterface {    #A
      name,
      ingredients {    #B
        ingredient {    #C
          name
        }
      }
    }
  }
}
```

#A We add an inline fragment on the `ProductInterface` type
#B We select `ProductInterface`'s `ingredients` property. Since `ingredients` points to the `IngredientRecipe` type, we must add a nested selector of properties from that type
#C We select `IngredientRecipe`'s `ingredient` property. `IngredientRecipe`'s `ingredient` property points to the `Ingredient` type, so we add another nested selector with `Ingredient`'s name property

Listing 9.10 leverages the connection between the `ProductInterface` and `Ingredient` types to fetch information from both types in a single query, but we can take this further. The `Ingredient` type contains a `supplier` property which point to the `Supplier` type. Say we want to get a list of products, including their names and ingredients, together with the supplier's name of each ingredient. Listing 9.11 shows how to do that (I encourage you to head over to the voyager UI to visualize the relationships captured by this query; head over to figure 9.1 for an illustration of the voyager UI).

9.11 Traversing the products API graph by leveraging the connections between types

```
{
  allProducts {
    ...on ProductInterface {    #A
      name
      ingredients {    #B
        ingredient {    #C
          name
          supplier {    #D
            name
          }
        }
      }
    }
  }
}
```

#A We add an inline fragment on the `ProudctInterface` type
#B We add a selector for `ProductInterface`'s `ingredients` property. ProductInterface's `ingredients` property points to the `IngredientRecipe` type, and therefore we include an inline fragment for `IngredientRecipe`
#C We add a selector for `IngredientRecipe`'s ingredient property. `IngredientRecipe`'s `ingredient` property points to the `Ingredient` type, and therefore we add an inline fragment for the `Ingredient` type
#D We add a selector for `Ingredient`'s `supplier` property. `Ingredient`'s `supplier` property points to the `Supplier` type, and therefore we add an inline fragment for the `Supplier` type

What listing 9.11 is doing is traversing our graph of types. Starting from the `ProductInterface` type, we are able to fetch details about other objects, such as `Ingredient` and `Supplier`, by leveraging their connections.

Here lies one the most powerful features of GraphQL, and one of its main advantages in comparison with other types of APIs, such as REST. Using REST, we'd need to make multiple requests to obtain all the information we were able to fetch in one request in listing 9.11. GraphQL gives you the power to obtain all the information you need, and just the information you need, in a single request.

Now that we know how traverse the graph of types in a GraphQL API, let's take our querying skills to the next level by learning how to run multiple queries within a single request!

9.6 Running multiple queries and query aliasing

This section explains how to run multiple queries per request, and how to create aliases for the responses returned by the server. Aliasing our queries means changing the key under which the dataset returned by the server is indexed. As we'll see, aliases can improve the readability of the results returned by the server, especially when we make multiple queries per request.

9.6.1 Running multiple queries in the same request

In previous sections, we run only one query per request. However, GraphQL also allows us to send several queries in one request. This is yet another powerful feature of GraphQL that can help us save unnecessary network roundtrips to the server, improving the overall performance of our applications and therefore user experience.

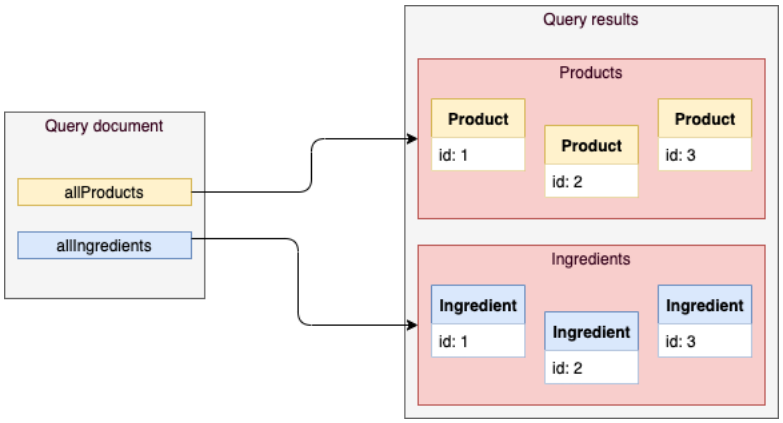


Figure 9.4 In GraphQL, we can run multiple queries within the same request, and the response will contain one dataset for each query.

Let's say we wanted to obtain a list of all the products and ingredients available in the CoffeeMesh platform, as shown in figure 9.4. To do that, we can run the `allIngredients` together with the `allProducts` queries. Listing 9.12 shows how we include both operations within the same query document. By including multiple queries within the same query document, we make sure that all of them are send to the server in the same request, and therefore we save roundtrips to the server.

9.12 Multiple queries per request

```
{
  allProducts {    #A
    ...commonProperties    #B
  }
  allIngredients {    #C
    name
  }
}

fragment commonProperties on ProductInterface {    #D
  name
}
```

#A We run the `allProducts` query without parameters
#B We include a named fragment to select properties from products
#C We run the `allIngredients` query within the same query document
#D We add a named fragment to select properties on the `ProductInterface` type

9.6.2 Aliasing our queries

All the queries we've run so far in previous sections are anonymous queries. When we make an anonymous query, the data returned by the server appears under a key named after the name of the query we're calling, as shown in listing 9.13.

9.13 Result of an anonymous query

```
# Query:
{
  allIngredients {    #A
    name
  }
}

# Result:
{
  "data": {    #B
    "allIngredients": [    #C
      {
        "name": "string"
      },
      {
        "name": "string"
      }
    ]
  }
}
```

#A We run the `allIngredients` query
#B The query returns successfully, so the returned payload contains a `data` key
#C Under `data`, we find the result of the query under a key named after the query itself

Running anonymous queries can sometimes be confusing. The `allIngredients` returns a list of ingredients, so it would be helpful to index the list of ingredients under an `ingredients` key, instead of `allIngredients`. Changing the name of this key is called **query aliasing**. We can make our queries more readable by using aliasing. The benefits of aliasing become clearer when we include multiple queries in the same request. For example, the query for all products and ingredients shown in listing 9.12 becomes more readable if we use aliases. Listing 9.14 how we use aliases to rename the results of each query: the result of `allProducts` appears under the `product` key, and the result of the `allIngredients` query appears under the `ingredients` key.

9.14 Using query aliasing for more readable queries

```
{
  products: allProducts {    #A
    ...commonProperties    #B
  }
  ingredients: allIngredients {    #C
    name
  }
}

fragment commonProperties on ProductInterface {    #D
  name
}
```

#A We add an alias called `products` to capture the result of the `allProducts` query
#B We use the named fragment `commonProperties` to select properties from the products
#C We add an alias called `ingredients` to capture the result of the `allIngredients` query
#D We add a named fragment to select properties on the `ProductInterface`

In some cases, using query aliases is necessary to make our requests work. For example, in listing 9.15 we ask the API for two datasets: one for available products and another one for unavailable products. Both datasets are produced by the same query: `products`. As you can see in listing 9.15, without query aliasing, this request results in conflict error, because both datasets would return under the same key: products.

9.15 Calling the same query multiple times without aliases causes an error

```
{
  products(input: {available: true}) {    #A
    ...commonProperties    #B
  }
  products(input: {available: false}) {    #C
    ...commonProperties
  }
}

fragment commonProperties on ProductInterface {    #D
  name
}
```

```
# Error
{
  "errors": [    #E
    {
      "message": "Fields \"products\" conflict because they have differing arguments. Use
       different aliases on the fields to fetch both if this was intentional.",    #F
      "locations": [
        {
          "line": 2,    #G
          "column": 3
        },
        {
          "line": 5,
          "column": 3
        }
      ]
    }
  ]
}
```

#A We run the products query filtering for available products
#B We use the commonPropetries fragment to select product properties
#C We run the products query again, this time filtering for unavailable products
#D We define the commonProperties fragment to select properties on the ProductInterface type
#E The query returns an unsuccessful response, so the payload includes an error key
#F The error message says that the query document contains a conflict
#G The server found errors in lines 2 and 5 from the query document

To resolve the conflict created by the queries in listing 9.15, we must use aliases. Listing 9.16 fixes the query by adding an alias to each operation: `availableProducts` for the query that filters for available products, and `unavailableProducts` for the query that filters for unavailable products.

9.16 Calling the same query multiple times with aliases

```
{
  availableProducts: products(input: {available: true}) {    #A
    ...commonProperties
  }
  unavailableProducts: products(input: {available: false}) {    #B
    ...commonProperties
  }
}

# Result (datasets ommitted for brevity)
{
  "data": {    #C
    "availableProducts": [...],    #D
    "unavailableProducts": [...]    #E
  }
}
```

#A We add the alias `availableProducts` for the query for available products
#B We add the alias `unavailableProducts` for the query for unavailable products
#C The query returns successfully, so the return payload contains a `data` keyword

#D The available products dataset appears under the `availableProducts` key
#E The unavailable products dataset appears under the `unavailableProducts` key

This concludes our overview of GraphQL queries. You've learned to run queries with parameters, with input types, with inline and named fragments, with aliases, and to include multiple queries within the same request. We've come a long way! But no overview of the GraphQL query language would be complete without learning how to run mutations. Let's now turn our attention to GraphQL mutations!

9.7 Running GraphQL mutations

This section explains how we run GraphQL mutations mutations. Mutations are GraphQL functions that allow us to create resources or change the state of the server. Running a mutation is similar to running a query. The only difference between the two is in their intent: queries are meant to read data from the server, while mutations are meant to create or change data in the server.

Let's illustrate how we run a mutation this with an example. Listing 9.17 shows how we run the `deleteProduct` mutation. When we use mutations, we must start our query document by qualifying our operation as a mutation. The `deleteProduct` mutation has one required argument: a product ID, and its return value is a simple Boolean, so in this case we don't have to include a selection set.

9.17 Calling a mutation

```
mutation {     #A
  deleteProduct(id: 'asdf')     #B
}
```

#A We qualify the operation we're going to run as a mutation
#B We call the `deleteProduct` mutation passing in the required id parameter

Let's now look at a more complex mutation, like `addProduct`, which is used to add new prodcuts to the CoffeeMesh catalogue. `addProduct` has three required parameters:

- `name`: the product name
- `type`: the product type. The values for this parameter are constrined by the `ProductType` enumeration, which offers two choices: cake and beverage.
- `input`: additional product properties, such as its price, size, list of ingredients, and others. The full list of properties is given by the `AddProductInput` type.

`addProduct` returns a value of type `Product`, which means in this case we must include a selection set. Remember that `Product` is the union of the `Cake` and the `Beverage` types, so our selection set must use fragments to indicate which type's properties we want to include in our return payload. Listing 9.18 shows how we run the `addProduct` mutation. In our example, we're selecting the `name` property on the `ProductInterface` type.

9.18 Calling a mutation with input parameters and complex return type

```
mutation {    #A
  addProduct(name: "Moccha", type: beverage, input: {price: 10, size: BIG, ingredients: [1,
       2]}) {    #B
    ...commonProperties    #C
  }
}

fragment commonProperties on ProductInterface {
  name
}
```

#A We qualify the operation we're going to run as a mutation
#B We call the `addProduct` mutation with all the required parameters: `name`, `type`, and `input`
#C We use a named fragment to select properties from `addProduct`'s return payload

Now that we know how to run mutations, it's time to learn how we write more structure and readable query documents by using parametrize the arguments. That'll be the topic of the next section!

9.8 Running parametrized queries and mutations

This section introduces parametrized queries and explains how we can use them to build more structure and readable query documents. In previous sections, when using queries and mutations that require parameters, we've defined the values for each parameter in the same line where we call the function. In queries with lots of arguments, this approach can lead to query documents which are cluttered and difficult to read and maintain. GraphQL offers a solution for this, which is to use parametrized queries.

9.5 GraphiQL offers a Query Variables panel where we can include the input values for our parametrized queries.

Parametrized queries allow us to decouple our query/mutation calls from the data. Figure 9.5 illustrates how we parametrize the call to the `addProduct` mutation using GraphiQL (the code for the query is also shown in listing 9.19 so that you can inspect it and copy it more easily). There're two things we need to do when we parametrize a query or mutation: 1) create a function wrapper around the query/mutation, and 2) assign values for the query/mutation parameters in a query variables object. Figure 9.6 illustrates how all these pieces fit together to bind the parametrized values to the `addProduct` mutation call.

9.19 Using parametrized syntax

```
# Query document
mutation CreateProduct($name: String!, $type: ProductType!, $input: AddProductInput!) {
       #A
  addProduct(name: $name, type: $type, input: $input) {    #B
    ...commonProperties    #C
  }
}

fragment commonProperties on ProductInterface {
  name
}

# Query variables
{
  "name": "Moccha",    #D
  "type": "beverage",    #E
  "input": {    #F
    "price": 10,
    "size": "BIG",
    "ingredients": [1, 2]
  }
}
```

#A We declare the operation as a mutation and we define the function wrapper for parametrizing the arguments. The wrapper's signature specifies the expected type of each argument
#B We call the `addProduct` mutation, passing in the parametrized arguments
#C We use a named fragment to select properties on the object returned by the `addProduct` mutation
#D We assign a value to the `name` parameter
#E We assign a value to the `type` parameter
#F We assign a value to the `input` parameter. The properties we can set in input are given by the `AddProductInput` type

Let's look at each of these steps in detail.

1. Creating a query/mutation wrapper. To parametrize our queries, we create a function wrapper around the query or mutation. In figure 9.5, we call the wrapper `CreateProduct`. The syntax for the wrapper looks very similar to the syntax we use to define a query. Parametrized arguments must be included in the wrapper's function signature. In figure 9.5, we parametrize the `name`, `type`, and `input` parameters of the `addProduct` mutation. The parametrized argument is marked with a dollar sign ($). In the wrapper's signature (i.e. `CreateProduct`), we specify the expected type of the parametrized arguments.

2. Parametrizing through a query variables object. Separately, we define our query variables as a JSON document. As you can see in figure 9.5, in GraphiQL we define query variables within the "Query Variables" panel. For further clarification on how parametrized queries work, have a look at figure 9.13.

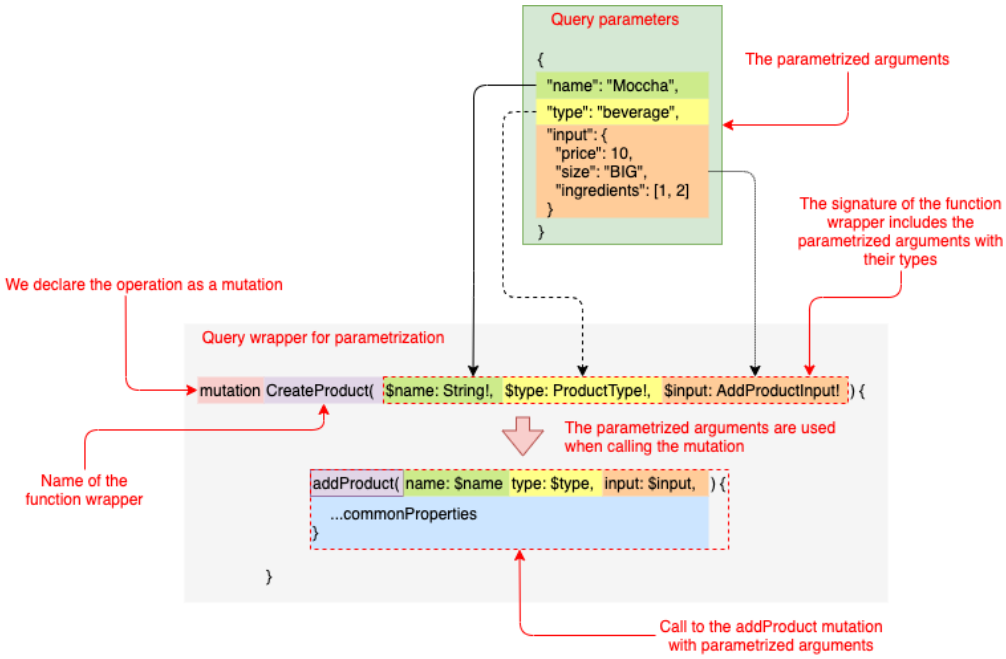


Figure 9.6 To parametrize queries and mutations, we create a function wrapper around the query or mutation. In the wrapper's signature we include the parametrized arguments. Parametrized variables carry a leading dollar ($) sign.

In figure 9.5 we used parametrized syntax to wrap only one mutation, but nothing prevents us from wrapping more mutations within the same query document. When we wrap multiple queries or mutations, all the parametrized arguments must be defined within the wrapper's function signature. Listing 9.20 shows how we extend the query from listing 9.19 to include a call to the `deleteProduct` mutation. In listing 9.20, we call the wrapper `CreateAndDeleteProduct` to better represent the actions in this request.

9.20 Using parametrized syntax

```
# Query document
mutation CreateAndDeleteProduct($name: String!, $type: ProductType!, $input:
       AddProductInput!, $id: ID) {    #A
  addProduct(name: $name, type: $type, input: $input) {    #B
    ...commonProperties    #C
  }
  deleteProduct(id: $id)    #D
}

fragment commonProperties on ProductInterface {
  name
}

# Query variables
{
  "name": "Moccha",    #E
  "type": "beverage",
  "input": {
    "price": 10,
    "size": "BIG",
    "ingredients": [1, 2]
  },
  "id": "asdf"    #F
}
```

#A We declare the operation as a mutation and we define the function wrapper for parametrizing arguments. The wrapper's signature includes the type of all the argument we wish to parametrize
#B We call the `addProduct` mutation passing in the parametrized arguments
#C We use a named fragment to select properties on `addProduct`'s return type
#D We call the `deleteProduct` mutation passing in the parametrized argument
#E We assign values to `addProduct`'s parameters: `name`, `type`, and `input`
#F We set the value for `deleteProduct`'s `id` parameter within the same payload, since both mutations are included within the same wrapper

This completes our journey to learn how to consume GraphQL APIs. You're now in a position to inspect any GraphQL API, explore its types, and play around with its queries and mutations. Before we close this chapter, I'd like to show you how a GraphQL API request actually works under the hood. That's the topic of the next section.

9.9 Demystifying GraphQL queries

This section explains how GraphQL queries work under the hood in the context of HTTP requests. In previous sections, we've used the GraphiQL client to explore our GraphQL API and to interact with it. GraphiQL translates our query documents into HTTP requests that the GraphQL server understands. GraphQL clients such as GraphiQL are interfaces that make it easier to interact with a GraphQL API. But nothing prevents you from sending an HTTP request directly to the API, say from your terminal, using something like cURL. Contrary to a popular misconception, you don't really need any special tools to work with GraphQL APIs[1].

[1] Unless you want to use subscriptions. Subscriptions are connections with the GraphQL server that allow you to receive notifications when something happens in the server, for example, when the state of a resource changes. Subscriptions require a two-way connection with the server, so you need something more sophisticated than cURL. To learn more about GraphQL subscriptions, see Eve Porcello and Alex Banks, *Learning GraphQL, Declarative Data Fetching for Modern Web Apps*, (O'Reilly, 2018), pp. 50-53 and 150-160.

To send a request to a GraphQL API, you can use either the GET or the POST methods. If you use GET, you send your query document using URL query parameters, and if you use POST, you include the query in the request payload. Graphql-faker's mock server only accepts GET requests, so I'll illustrate how you send a query using GET.

Let's run the `allIngredients` query and selecting only the name property of each ingredient. Since this is a GET request, our query document must be included in the URL as a query parameter. However, the query document contains special characters, such as curly braces, which are considered unsafe and therefore cannot be included in a URL. To deal with special characters in URLs, we URL encode them. URL encoding is the process of translating special characters, such as braces, punctuation marks, and others, into a suitable format for URLs. URL encoded characters start with a percent sign, so this type of encoding is also known as percent encoding[2]. cURL takes care of URL encoding our data when we use the `--data-urlencode` option. By using `--data-urlencode`, cURL translates our command into a GET request with the following URL: http://localhost:9002/graphql?query=%7BallIngredients%7Bname%7D%7D. The following snippet shows the cURL command you need to run to make this call:

```
curl 'http://localhost:9002/graphql' --data-urlencode 'query={allIngredients{name}}'
```

Now that you understand how GraphQL API requests work under the hood, let's see how we can leverage this knowledge to write code in Python that consumes a GraphQL API. That'll be the topic of the next section!

9.10 Calling a GraphQL API with Python code

This section illustrates how we can interact with a GraphQL API using Python. GraphQL clients like GraphiQL are useful to explore and get familiar with a GraphQL API, but in practice, you'll spend most of your time writing applications that consume those APIs programmatically. In this section, we learn to consume the products API using a GraphQL client written in Python.

To work with GraphQL APIs, the Python ecosystem counts with libraries such as gql (https://github.com/graphql-python/gql) and sgqlc (https://github.com/profusion/sgqlc). These libraries are useful when we want to use advanced features of GraphQL, such as subscriptions. You'll rarely need those features in the context of microservices, so for the purposes of this section, we'll take a simpler approach and use the popular requests library (https://github.com/psf/requests). Remember that GraphQL queries are simply GET or POST requests with a query document.

Listing 9.21 shows how we call the `allIngredients` query, adding a selector for `Ingredient`'s `name` property. Since our GraphQL mock server only accepts GET requests, we send the query document in the form of URL encoded data. With requests, we accomplish this by passing the query document through the `get` method's `params` argument. As you can see, the query document looks the same as what we wrote in the GraphiQL query panel, and the result from the API also looks the same. This is great news, because it means that, when

[2] See the Wikipedia article about "Percent encoding": https://en.wikipedia.org/wiki/Percent-encoding [accessed 2021/02/28).

working out your queries, you can start working with GraphiQL, leveraging its great support for syntax highlighting and query validation, and when you're ready, you can move your queries directly to your Python code.

9.21 Calling a GraphQL query using Python

```
# Script
import requests    #A

URL = 'http://localhost:9002/graphql'    #B

query_document = '''    #C
{
  allIngredients {
    name
  }
}
'''

result = requests.get(URL, params={'query': query_document})    #D

result = result.json()      #E

# Result
{'data': {'allIngredients': [{'name': 'string'}, {'name': 'string'}, {'name': 'string'}]}}
```

#A We import the requests library
#B We declare a variable to hold the base URL of our GraphqL server
#C We declare a variable to hold the query document that we're going to send to the GraphQL server
#D We use the requests library to send a GET request to the server, passing in the query document as a URL query parameter
#E We parse the JSON payload returned by the server

This concludes our journey through GraphQL. You went from learning about the basic scalar types supported by GraphQL in chapter 8, to making complex queries using tools as varied as GraphiQL, cURL, and Python in this chapter. Along the way, we built the specification for the products API, and we interacted with it using a GraphQL mock server. That's no small feat. If you've read so far, you've learned a great deal of things about APIs and you should be proud of it!

GraphQL is one of the most popular protocols in the world of web APIs, and its adoption grows every year. GraphQL is a great choice for building microservices APIs and for integration with frontend applications. In the next chapter, we'll undertake the actual implementation of the products API and its service. Stay tuned!

9.11 Summary

- When we call a query or mutation that returns an object type, our query must include a selection set. A selection set is a list of the properties that we want to fetch from the object returned by the query.
- When a query or mutation returns a list of multiple types, our selection set must include fragments. Fragments are selections of properties on a specific type, and they're prefixed by the spread operator (three dots).
- When calling a query or mutation that includes arguments, we can parametrize those arguments by building a wrapper around the query or queries. This allows us to write more readable and maintainable query documents.
- When designing a GraphQL API, it's a good idea to put it to work with a mock server. Mock servers allow us to build API clients while the server is implemented.
- You can run a GraphQL mock server using graphql-faker, which also creates a GraphiQL interface to the API. This is useful to test that our design conforms to our expectations.
- Behind the scenes, a GraphQL query is a simple HTTP request which uses either the GET or the POST methods. When using GET, we must ensure our query document is URL encoded, and when using POST, we include it in the request payload.

10

Implementing a GraphQL API

This chapter covers

- Creating GraphQL APIs using the Ariadne web server framework
- Validating request and response payloads
- Creating resolvers for queries and mutations
- Creating resolvers for complex object types, such as union types
- Creating resolvers for custom scalar types
- Creating resolvers for object properties

In chapter 8, we designed a GraphQL API for the products service, and we produced a specification detailing the requirements for the products API. In this chapter, we'll implement the API according to the specification. To build the API, we'll use the Ariadne framework, which is one of the most popular GraphQL libraries in the Python ecosystem. Ariadne allows us to leverage the benefits of documentation-driven development by automatically loading data validation models from the specification. We'll learn to create resolvers, which are Python functions that implement the logic of a query or mutation. We'll also learn to handle queries that return multiple types. After reading this chapter, you'll have all the tools you need under your belt to start developing your own GraphQL APIs!

The code for this chapter is available in the GitHub repository provided with this book, under the folder ch09. Unless otherwise specified, all the file references within this chapter are relative to the ch09 folder. For example, server.py refers to the ch09/server.py file, and web/schema.py refers to the ch09/web/schema.py file. Also, to ensure all the commands used in this chapter work as expected, make sure you `cd` into the ch09 folder in your terminal.

10.1 Analyzing the API requirements

In this section, we analyze the requirements of the API specification. Before jumping on to implement an API, it's worth spending some time analyzing the API specification and what it requires. Let's do this analysis for the products API!

The products API specification is available in the ch09/web/products.graphql file, in the repository provided with this book. The specification defines a collection of object types that represent the data that we can retrieve from the API, and a set of queries and mutations that expose the capabilities of the products service. We have to create validation models that faithfully represent the schemas defined in the specification, as well as functions that correctly implement the functionality of the queries and mutations.

As we comment in the next section, we'll work with a framework that can handle schema validation automatically from the specification, so we don't need to worry about implementing validation models.

Our implementation will focus mainly on the queries and mutations. Most of the queries and mutations defined in the schema return either an array or a single instance of the `Ingredient` and the `Product` types. `Ingredient` is simpler since it's an object type, so we'll look at queries and mutations that use this type first. `Product` is the union of the `Beverage` and the `Cake` types, both of which implement the `ProductInterface` type. As we'll see, implementing queries and mutations that return union types is slightly more complex. A query that returns a list of `Product` objects contains instances of both the `Beverage` and the `Cake` types, so we need to implement additional functionality that makes it possible for the server to determine which type each element in the list belongs to.

With that said, let's analyze the tech stack that we'll use for this chapter, and then move straight into the implementation!

10.2 Introducing the tech stack

In this section, we discuss the tech stack that we'll use to implement the products API. We discuss which libraries are available for implementing GraphQL APIs in Python, and we choose one of them. We also discuss the server framework that we'll use to run the application.

Since we're going to implement a GraphQL API, the first thing we want to look for is a good GraphQL server library. GraphQL's website (https://graphql.org/code/) is an excellent resource for finding tools and frameworks for the GraphQL ecosystem. As the ecosystem is constantly evolving, I recommend you check out that website every once in a while for any new additions. The website lists four Python libraries that support GraphQL:

- **Graphene** (https://github.com/graphql-python/graphene) is one of the first GraphQL libraries built for Python. It's battle-tested and one of the most widely used libraries.
- **Ariadne** (https://github.com/mirumee/ariadne) is a library built for schema-first (or documentation-driven) development. It's a highly popular framework and it handles schema validation and serialization automatically.
- **Strawberry** (https://github.com/strawberry-graphql/strawberry) is a more recent library which makes it really easy to implement GraphQL schema models by offering a clean interface inspired by Python dataclasses.
- **Tartiflette** (https://github.com/tartiflette/tartiflette) is another recent addition to the Python ecosystem which allows you to implement a GraphQL server using a schema-first approach, and it's built on top of asyncio, which is Python's core library for asynchronous programming.

For this chapter, we'll use Ariadne, since it supports a schema-first or documentation-driven development approach and it's a mature project. Since we already have the API specification available, we don't want to spend time implementing each schema model in Python. Instead, we want to use a library that can handle schema validation and serialization directly from the API specification, and Ariadne can do that.

We'll run the Ariadne server with the help of uvicorn, which we encountered in chapters 2 and 6 when we worked with FastAPI. To install the dependencies for this chapter, you can use the Pipfile and Pipfile.lock files available under the ch09 folder in the repository provided with this book. Copy the Pipfile and Pipfile.lock files into your ch09 folder, `cd` into it, and run the following command:

```
pipenv install
```

If you prefer to install the latest versions of Ariadne and uvicorn, simply run:

```
pipenv install ariadne uvicorn
```

Now that we have the dependencies installed, let's activate the environment:

```
pipenv shell
```

With all the dependencies installed, now we are ready to start coding, so let's do it!

10.3 Introducing Ariadne

In this section, we introduce the Ariadne framework, and we learn how it works by using a simple example. We'll learn how to run a GraphQL server with Ariadne, how to load a GraphQL specification, and how to implement a simple GraphQL resolver. As we saw in chapter 9, users interact with GraphQL APIs by running queries and mutations. A GraphQL resolver is a function that knows how to execute one of those queries or mutations. In our implementation, we'll have as many resolvers as queries and mutations there are in the API specification. As you can see from figure 10.1, resolvers are the pillars of a GraphQL server, since it's through resolvers that we can return actual data to the users of the API.

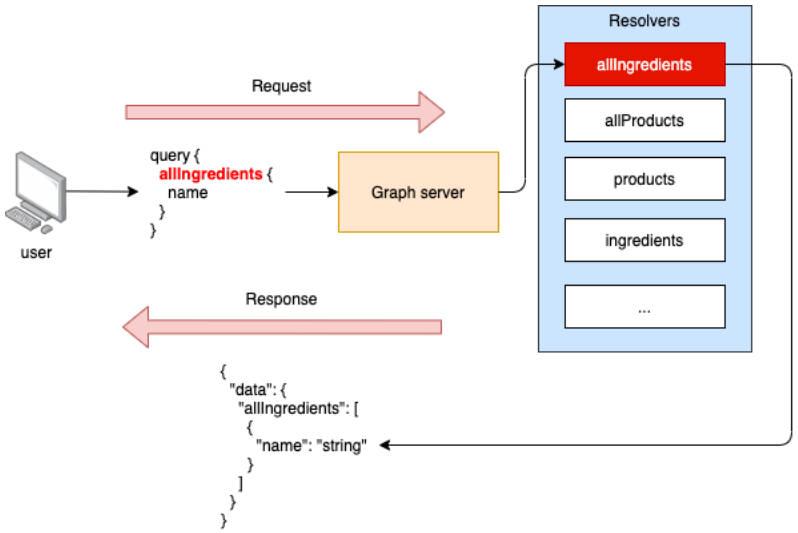


Figure 10.1 To serve data to a user, a GraphQL server uses resolvers which know how to build the payload for a given query.

Let's start by writing a very simple GraphQL schema. Open the ch09/server.py file and copy the following content into it:

```
schema = '''
  type Query {
    hello: String
  }
'''
```

We define a variable called `schema`, and we point it to a simple GraphQL schema. This schema defines only one query, named `hello`, which returns a string. The return value of the `hello` query is optional, which means `null` is also a valid return value. To expose this query through our GraphQL server, we need to implement a resolver using Ariadne.

Ariadne can run a GraphQL server from this simple schema definition. How do we do that? First, we need to load the schema using Ariadne's `make_executable_schema` function. `make_executable_schema` parses the document, validates our definitions, and builds an internal representation of the schema. As you can see in figure 10.2, Ariadne uses the output of this function to validate our data. For example, when we return the payload for a query, Ariadne validates the payload against the schema.

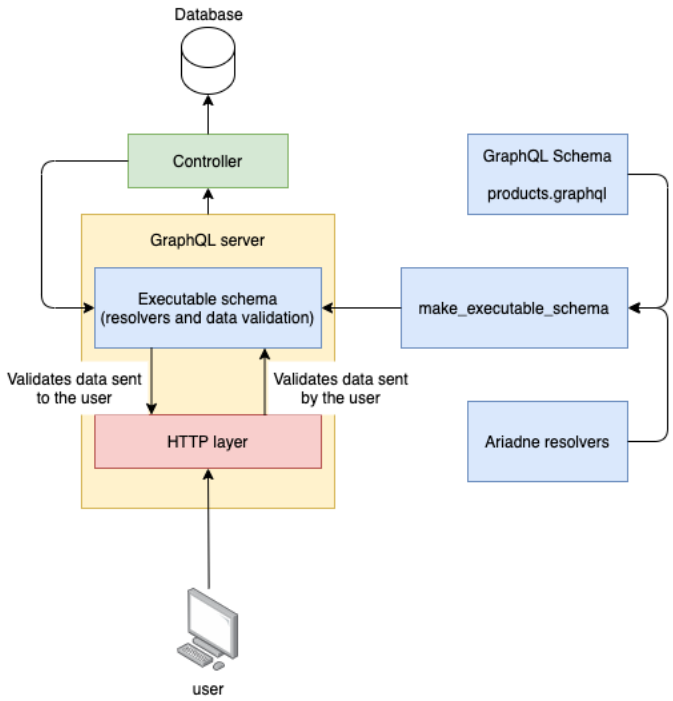


Figure 10.2 To run the GraphQL server with Ariadne, we produce an executable schema by loading the GraphQL schema for the API and a collection of resolvers for the queries and mutations. Ariadne uses the executable schema to validate data sent by the user to the server, as well as data sent from the server to the user.

Once we've loaded the schema, we can initialize our server using Ariadne's `GraphQL` class. Ariadne provides two implementations of the server: a synchronous implementation, which is available under the `ariande.wsgi` module, and an asynchronous implementation, which is available under the `ariande.asgi` module[1]. In this chapter, we'll use the asynchronous implementation. Listing 10.1 shows how all this comes together in ch09/server.py.

[1] For an explanation of the differences between synchronous and asynchronous servers, head over to the sidebar titled "Asynchronous servers and performance" in section 6.6.

Listing 10.1 Initializing a GraphQL server using Ariadne

```
from ariadne import make_executable_schema
from ariadne.asgi import GraphQL

schema = '''
  type Query {    #A
    hello: String
  }
'''

server = GraphQL(make_executable_schema(schema), debug=True)    #B
```

#A We declare a simple schema with a single query, hello, which returns a string
#B We create the server by instantiating the GraphQL class and passing it the schema for the GraphQL API

To run the server, execute the following command from the terminal:

```
$ uvicorn server:server --reload
```

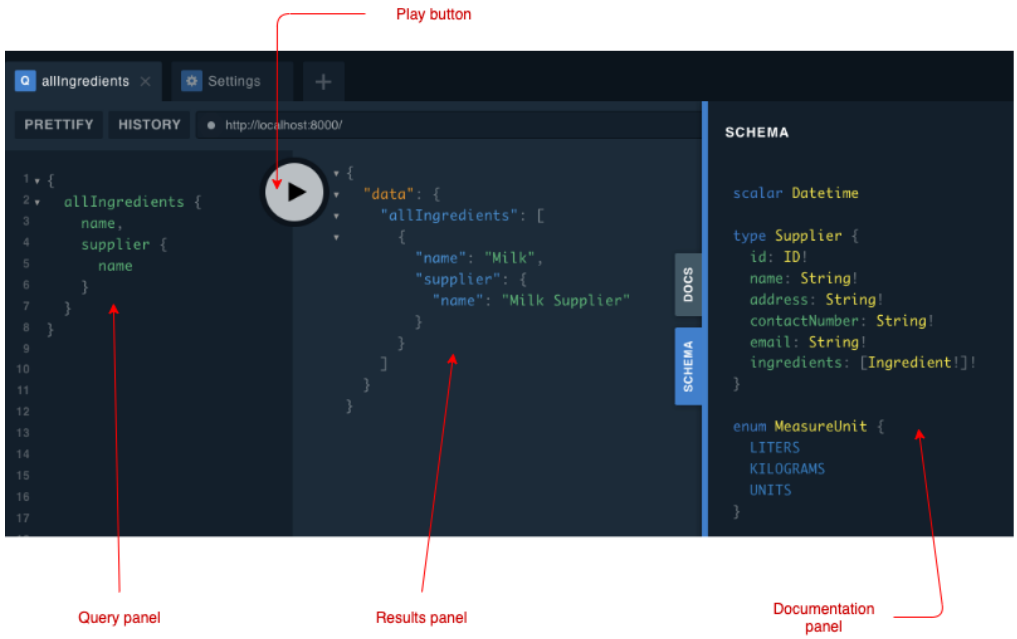


Figure 10.3 The Apollo Playground interface contains a query panel where we execute queries and mutations; a results panel where the queries and mutations are evaluated; and a documentation panel where we can inspect the API schemas.

Your application will be available on http://localhost:8000. If you head over to that address, you'll see an Apollo Playground interface to the application. As you can see in figure

10.3, Apollo Playground is similar to GraphiQL, which we learned in chapter 8. On the left side panel, we write our queries. Go ahead and write the following query:

```
{
  hello
}
```

This query executes the query function that we defined in listing 10.1 If you hit the execute button, you'll get the results of this query on the right-side panel:

```
{
  "data": {
    "hello": null
  }
}
```

The query returns `null`. This shouldn't come as a surprise, since the return value of the hello query is a nullable string. How can we make the `hello` query return a string? Enter **resolvers**. Resolvers are functions that let the server know how to produce a value for a type or an attribute. To make the `hello` query return an actual string, we need to implement a resolver. Let's create a resolver that returns a string of ten random characters.

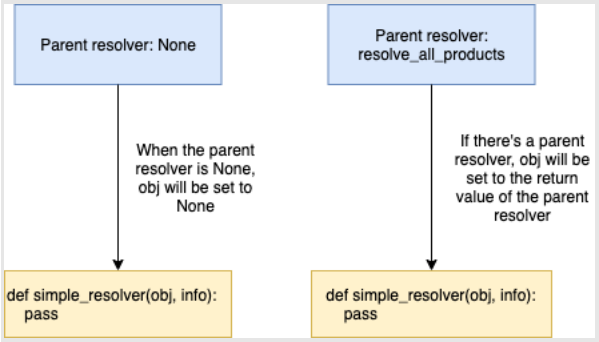
In Ariadne, a resolver is a Python callable (for example, a function) that takes two positional parameters: `obj` and `info`. To learn more about these resolvers, head over to the sidebar titled Resolver parameters.

Resolver parameters in Ariadne

Ariadne's resolvers always have two positional-only parameters, which are commonly called `obj` and `info`. The signature of a basic Ariadne resolver is

```
def simple_resolver(obj: Any, info: GraphQLResolveInfo):
  pass
```

As you can see in the figure of this sidebar, `obj` will normally be set to `None`, unless the resolver has a parent resolver, in which case `obj` will be set to the value returned by the parent resolver. We encounter the latter case when a resolver doesn't return an explicit type. For example, the resolver for the `allProducts` query, which we'll implement in section 10.4.4, doesn't return an explicit type. It returns an object of type `Product`, which is the union of the `Cake` and `Beverage` types. To determine the type of each object, Ariadne needs to call a resolver for the `Product` type.



When a resolver doesn't have a parent resolver, the obj parameter is set to None. When there's a parent resolver, obj will be set to the value returned by the parent resolver.

The `info` parameter is an instance of `GraphQLResolveInfo`, which contains information required to execute a query. Ariadne uses this information to process and serve each request. For the application developer, the most interesting attribute exposed by the `info` object is `info.context`, which contains details about the context in which the resolver is called, such as the HTTP context. To learn more about the obj and info objects, check out Ariadne's documentation: https://ariadnegraphql.org/docs/resolvers.html.

A resolver needs to be bound to its corresponding object type. Ariadne provides bindable classes for each GraphQL type:

- `ObjectType` for object types.
- `QueryType` for query types. In GraphQL, the query type represents the collection of all queries available in a schema. As we saw in section 8.XXX, a query is a function that reads data from a GraphQL server.
- `MutationType` for mutation types. As we saw in section 8.XXX, a mutation is a function which alters the state of the GraphQL server.
- `UnionType` for union types.
- `InterfaceType` for interface types.
- `EnumType` for enumeration types.

Since `hello` is a query, we need to bind its resolver to an instance of Ariadne's `QueryType`. Listing 10.2 shows how we do that. We first create an instance of the `QueryType` class and assign it to a variable called `query`. We then use `QueryType`'s `field` decorator method to bind our resolver. The `field` decorator method is available on most of Ariadne's bindable classes, and it allows us to bind a resolver to a specific field. By convention, we prefix our resolvers' names with `resolve_`. Ariadne's resolvers always get two positional-only parameters by default: `obj` and `info`. We don't need to make use of those parameters in this case, so we use the expression `*_`, which is a convention in Python to ignore a list of positional parameters. To make Ariadne aware of our resolvers, we need to pass our bindable objects as an array to the `make_executable_schema` function. The changes in listing 10.2 go under ch09/server.py.

Listing 10.2 Implementing a GraphQL resolver with Ariadne

```
import random
import string

from ariadne import QueryType, make_executable_schema
from ariadne.asgi import GraphQL


query = QueryType()    #A


@query.field('hello')    #B
def resolve_hello(*_):    #C
    return ''.join(random.choice(string.ascii_letters) for _ in range(10))    #D


schema = '''
type Query {    #E
        hello: String
    }
'''

server = GraphQL(make_executable_schema(schema, [query]), debug=True)    #F
```

#A We create an instance of the QueryType bindable class so that we can bind query resolvers to it
#B We bind a resolver for the hello query using QueryType's field decorator
#C We skip Ariadne's resolvers default positional-only parameters since we don't need them
#D We return a list of randomly generated ascii characters
#E We declare a simple GraphQL schema with a single query called hello
#F We create the server by instantiating the GraphQL class, and passing it the executable schema for the GraphQL API, as well as an array of bindable classes

Since we're running the server with the hot reloading flag (`--reload`), the server automatically reloads once you save the changes to the file. Go back to the Apollo Playground interface in http://127.0.0.1:8000, and run the hello `query` again. This time, you should get a random string of ten characters as a result.

This completes our introduction to Ariadne. You've learned how to load a GraphQL schema with Ariadne, how to run the GraphQL server, and how to implement a resolver for a query function. In the rest of the chapter, we'll apply this knowledge as we build the GraphQL API for the products service.

10.4 Implementing the products API

In this section, we'll use everything we learned in the previous section to build the GraphQL API for the products service. In particular, you'll learn to build resolvers for the queries and mutations of the products API, to handle query parameters, and to structure your project. Along the way, we'll learn additional features of the Ariadne framework, and various strategies for testing and implementing GraphQL resolvers. By the end of this section, you'll be able to build GraphQL APIs for your own microservices. Let the journey begin!

10.4.1 Laying out the project structure

In this section, we structure our project for the products API implementation. So far, we've included all our code under the ch09/server.py file. To implement a whole API, we need to split our code into different files and add structure to the project, since otherwise the codebase would become difficult to read and to maintain. To keep the implementation simple, we'll use an in-memory representation of our data.

If you followed along with the code in the previous section, delete the code we wrote earlier under ch09/server.py. ch09/server.py represents the entry point to our application and therefore will contain an instance of the GraphQL server. We'll encapsulate the web server implementation within a folder called ch09/web. Go ahead and create such folder, and within it, create the following files:

- data.py will contain the in-memory representation of our data.
- mutations.py will contain resolvers for the mutations in the products API.
- queries.py will contain resolvers for queries.
- schema.py will contain all the code necessary to load an executable schema.
- types.py will contain resolvers for object types, custom scalar types, and object properties.

The products.graphql specification file also goes under the ch09/web folder, since it's handled by the code under the ch09/web/schema.py file. The directory structure for the products API looks like this:

```
.
├── Pipfile
├── Pipfile.lock
├── server.py
└── web
    ├── data.py
    ├── mutations.py
    ├── products.graphql
    ├── queries.py
    ├── schema.py
    └── types.py
```

Now that we have structured our project, it's time to start coding!

10.4.2 Creating an entry point for the GraphQL server

Now that we have structured our project, it's time to work on the implementation. In this section, we'll create the entry point for the GraphQL server. We need to create an instance of Ariadne's `GraphQL` class, and load an executable schema from the products specification.

As we mentioned in section 1.4.1, the entry point for the products API server lives under ch09/server.py. Go ahead and include the following content in this file:

```
from ariadne.asgi import GraphQL

from web.schema import schema

server = GraphQL(schema, debug=True)
```

Next, let's create the executable schema under ch09/web/schema.py:

```
from pathlib import Path

from ariadne import make_executable_schema

schema = make_executable_schema(
    (Path(__file__).parent / 'products.graphql').read_text()
)
```

The API specification for the products API is available under the ch09/web/products.graphql file. We read the schema file contents and pass them on to Ariadne's `make_executable_schema` function. We then pass the resulting schema object to Ariadne's `GraphQL` class to instantiate the server. You can start up now the server by running the following command:

```
$ uvicorn server:server --reload
```

Like before, the API is available on http://localhost:8000. If you visit this address again, you'll see the familiar Apollo Playground UI. At this point, we could try running any of the queries defined in the products API specification, however most of them will fail since we haven't implemented any resolvers yet. For example, if you run the following query:

```
 {
  allIngredients {
    name
  }
}
```

You'll get the following error message: "Cannot return null for non-nullable field Query.allProducts." The server doesn't know how to produce a value for the `Ingredient` type since we don't have a resolver for it. So, let's go ahead and build it!

10.4.3 Implementing query resolvers

In this section, we learn to implement query resolvers. As you can see from figure 10.4, a query resolver is a Python function that knows how to return a valid payload for a given query. We'll build a resolver for the `allIngredients` query, which is one of the simplest queries in the products API specification.

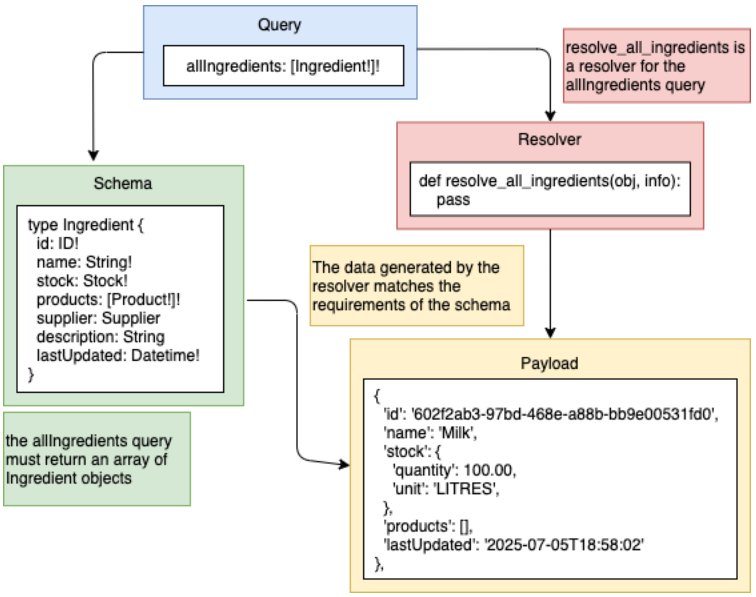


Figure 10.4 GraphQL uses resolvers to serve the queries requests sent by the user to the server. A resolver is a Python function that knows how to return a valid payload for a given query.

To implement a resolver for the `allIngredients` query, we simply need to create a function that returns a data structure with the shape of the `Ingredient` type. As you can see from listing 10.3, the `Ingredient` type has four non-nullable properties: `id`, `name`, `stock`, and `products`. The `stock` property is, in turn, an instance of the `Stock` object type, which, as per the specification, must contain the `quantity` and `unit` properties. Finally, the `products` property must be an array of `Product` objects. The contents of the array are non-nullable, but an empty array is a valid return value.

Listing 10.3 Specification for the Ingredient type

```
type Stock {    #A
    quantity: Float!    #B
    unit: MeasureUnit!
}

type Ingredient {
    id: ID!
    name: String!
    stock: Stock!
    products: [Product!]!    #C
    supplier: Supplier    #D
    description: String
    lastUpdated: Datetime!
}
```

#A GraphQL object types are declared with the type definition.
#B An exclamation mark trailing the definition of a field indicates that the field is non-nullable.
#C products is a non-nullable property, and its contents are non-nullable instances of the Product type.
#D When a field declaration doesn't have a tailing exclamation mark, it means it's nullable or non-required.

Let's add a list of ingredients to the in-memory list representation of our data under the web/data.py file:

```
# file: web/datap.py

ingredients = [
    {
        'id': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',
        'name': 'Milk',
        'stock': {
            'quantity': 100.00,
            'unit': 'LITRES',
        },
        'supplier': '92f2daae-a4f8-4aae-8d74-51dd74e5de6d',
        'products': [],
        'lastUpdated': datetime.now(),
    },
]
```

Now that we have some data, we can use it in the `allIngredient`'s resolver. Listing 10.4 shows what `allIngredients`'s resolver looks like. As we did in section 10.3, we first create an instance of the `QueryType` class, and we bind the resolver with this class. Since this is a resolver for a query type, the implementation goes under the web/queries.py file.

Listing 10.4 A resolver for the allIngredients query

```
# web/queries.py

from ariadne import QueryType

from web.data import ingredients

query = QueryType()


@query.field('allIngredients')    #A
def resolve_all_ingredients(*_):
    return ingredients    #B
```

#A We bind a resolver for the allIngredients query using the QueryType's field decorator
#B We return a hardcoded response with a payload that contains all the required attributes for the Ingredient type

To enable the query resolver, we have to pass the query object to the `make_executable_schema` function under web/schema.py:

```
schema = make_executable_schema(
    (Path(__file__).parent / 'products.graphql').read_text(), [query]
)
```

If we run the following query

```
{
  allIngredients {
    name
  }
}
```

We now get a valid payload. The query only selects the ingredient's name, which in itself is not very interesting, and it doesn't really tell us whether our current resolver works as expected for other fields. Let's write a more complex query to test our resolver more thoroughly. The following query selects the `id`, `name`, and `description` of an ingredient, as well as the `name` of each product it's related to:

```
{
  allIngredients {
    id,
    name,
    products {
      ...on ProdcutBase {
        name
      }
    },
    description
  }
}
```

The response payload to this query is also valid:

```
{
  "data": {
    "allIngredients": [
      {
        "id": " "602f2ab3-97bd-468e-a88b-bb9e00531fd0",
        "name": "Milk",
        "products": [],
        "description": null
      }
    ]
  }
}
```

The products list is empty because we haven't associated any products with the ingredient, and the `description` is `null` because this is a nullable field. Now that we know how to implement resolvers for simple queries, in the next section we'll learn to implement resolvers that handle more complex situations.

10.4.4 Implementing type resolvers

In this section, we'll learn to implement resolvers for queries that return multiple types. The `allIngredients` query is fairly simple since it only returns one type of object: the `Ingredient` type. Let's now consider the `allProducts` query. As you can see from figure 10.5, `allProducts` is more complex since it returns the `Product` type, which is a union of the `Beverage` and the `Cake` types, both of which implement the `ProductInterface` type.

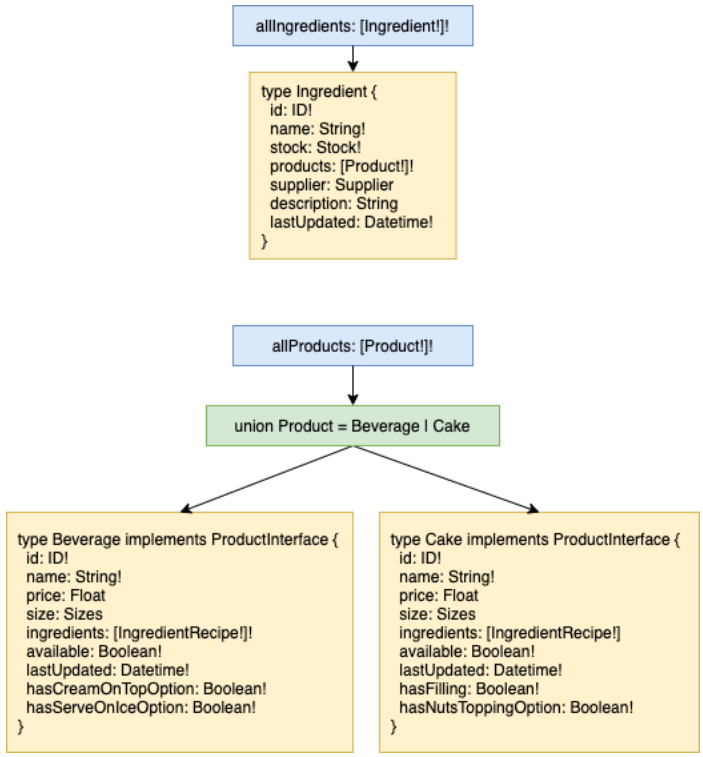


Figure 10.5 The allIngredients query returns an array of Ingredient objects, while the allProducts query returns an array of Product objects, where Product is the union of two types: Beverage and Cake.

Let's begin by adding a list of products to our in-memory list of data under the web/data.py file. We'll add two products: one `Beverage` and one `Cake`. What fields should we include in the products? As you can see in figure 10.6, since `Beverage` and `Cake` implement the `ProductInterface` type, we know they both require an `id`, a `name`, a list of `ingredients`, and a field called `available`, which signals whether the product is available. On top of these common fields inherited from `ProductInterface`, `Beverage` requires two additional fields: `hasCreamOnTopOption` and `hasServeOnIceOption`, both of which are Booleans. In turn, `Cake` requires the properties `hasFilling` and `hasNutsToppingOption`, both of which are also Booleans.

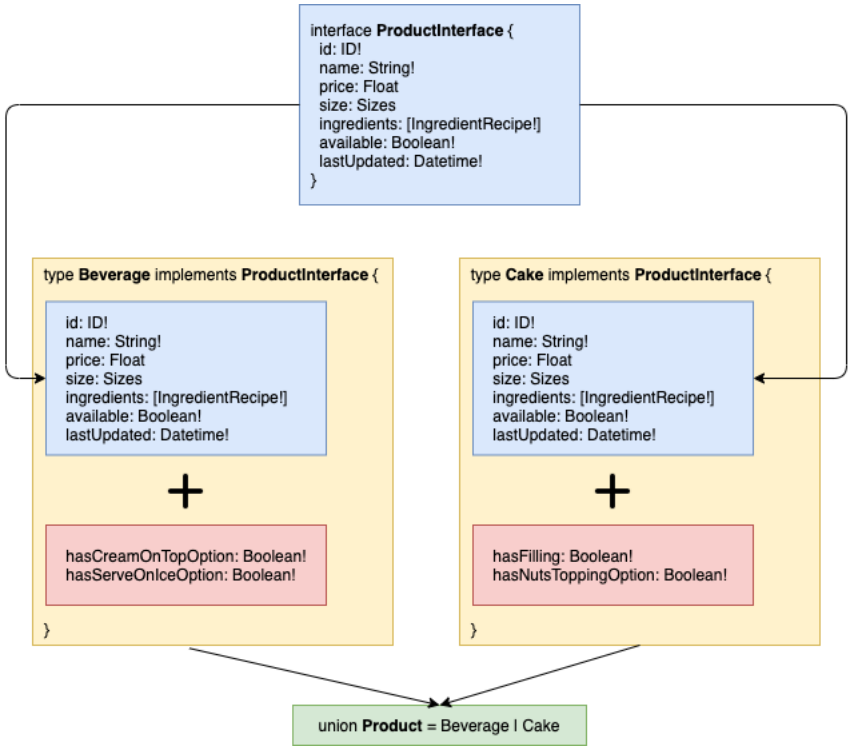


Figure 10.6 Product is the union of the Beverage and the Cake types, both of which implement the ProductInterface type. Since Beverage and Cake implement the same interface, both types share the properties inherited from the interface. In addition to those properties, each type has its own specific properties, such as hasFilling in the case of the Cake type.

Listing 10.5 shows the list of products under web/data.py.

Listing 10.5 Resolver for the allProducts query

```
# file: web/data.py

...

products = [
    {
        'id': '6961ca64-78f3-41d4-bc3b-a63550754bd8',
        'name': 'Walnut Bomb',
        'price': 37.00,
        'size': 'MEDIUM',
        'available': False,
        'ingredients': [
            {
                'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',    #A
                'quantity': 100.00,
                'unit': 'LITRES',
            }
        ],
        'hasFilling': False,
        'hasNutsToppingOption': True,
        'lastUpdated': datetime.now(),
    },
    {
        'id': 'e4e33d0b-1355-4735-9505-749e3fdf8a16',
        'name': 'Cappuccino Star',
        'price': 12.50,
        'size': 'SMALL',
        'available': True,
        'ingredients': [
            {
                'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',
                'quantity': 100.00,
                'unit': 'LITRES',
            }
        ],
        'hasCreamOnTopOption': True,
        'hasServeOnIceOption': True,
        'lastUpdated': datetime.now(),
    },
]
```

#A This ID references the ID of the milk ingredient we added earlier to web/data.py.

Now that we have a list of products, let's use it in the `allProducts`'s resolver. Listing 10.6 shows the changes we need to make to the web/queries.py to add the `allProducs` resolver.

Listing 10.6 Adding the allProducts resolver

```
# web/queries.py

from ariadne import QueryType

from web.data import ingredients, products

query = QueryType()

...


@query.field('allProducts')    #A
def resolve_all_products(*_):
    return products    #B
```

#A We bind a resolver for the allProducts query using QueryType's field decorator
#B We return a hardcoded payload of product objects

Let's run a simple query to test the resolver:

```
{
  allProducts {
    ...on ProdcutBase {
      name
    }
  }
}
```

If you run this query, you'll get an error saying that the server can't determine what types each of the elements in our list are. In these situations, we need a **type resolver**. As you can see in figure 10.7, a type resolver is a Python function that determines what type an object is, and it returns the name of the type.

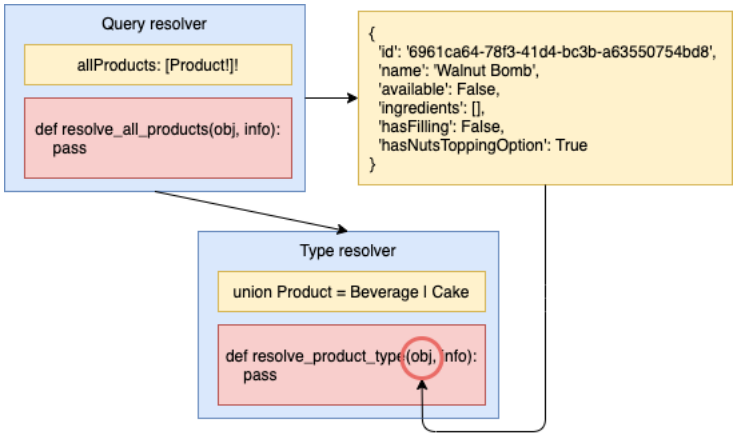


Figure 10.7 A type resolver is a function that determines the type of an object. This example shows how the resolve_product_type resolver determines the type of an object returned by the resolve_all_products resolver.

We need type resolvers in queries and mutations that return more than one object type. In the products API, this affects all queries and mutations that return the `Product` type, such as `allProducts`, `addProduct`, and `product`.

> **RETURNING MULTIPLE TYPES.** Whenever a query or mutation returns multiple types, you'll need to implement a type resolver. This applies to queries and mutations that return union types and object types that implement interfaces.

Listing 10.7 shows how we implement a type resolver for the `Product` type in Ariadne. The type resolver function takes two positional parameters, the first of which is an object. We need to determine the type of this object. As you can see in figure 10.8, since we know that `Cake` and `Beverage` have different required fields, we can use this information to determine their types: if the object has a `hasFilling` property, we know it's a `Cake`, otherwise it's a `Beverage`.

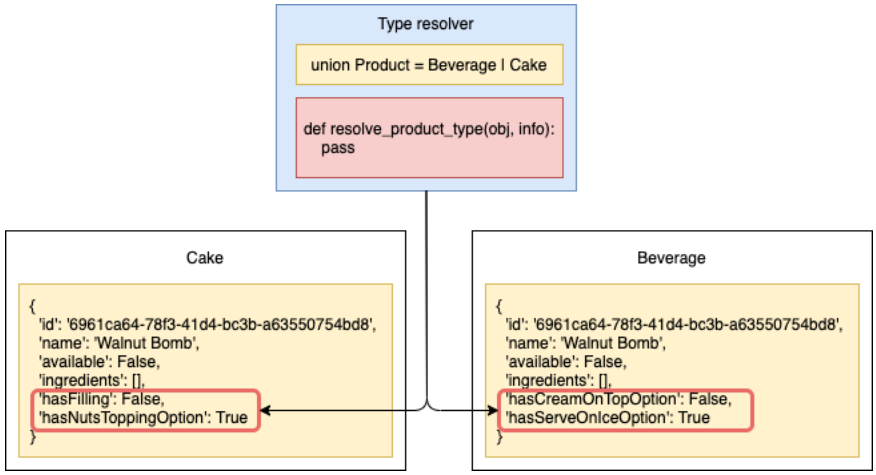


Figure 10.8 A type resolver inspects the properties of a payload to determine its type. In this example, resolve_product_type looks for distinguishiung properties that differentiate a Cake from a Beverage type.

The type resolver has to be bound to the `Product` type. Since `Product` is a union type, we create a bindable object of it using the `UnionType` class. Ariadne guarantees that the first argument in a resolver is an object, and we inspect this object to resolve its type. We don't need any other parameters, so we ignore them with Python's `*_` syntax, which is standard for ignoring positional parameters. To resolve the type of the object, we check whether it has a `hasFilling` attribute. If it does, we know it's a `Cake` object, otherwise it's a `Beverage`. Finally, we pass the product bindable to the `make_executable_schema` function. Since this is a type resolver, this code goes into the web/types.py.

Listing 10.7 Implementing a type resolver for the Product union type

```
# file: web/types.py

from ariadne import UnionType

product_type = UnionType('Product')    #A


@product_type.type_resolver    #B
def resolve_product_type(obj, *_):    #C
    if 'hasFilling' in obj:
        return 'Cake'
    return 'Beverage'
```

#A We create a bindable object for the Product type using the UnionType class
#B We bind a type resolver to Product by using UnionType's type_resolver decorator
#C We capture the first argument for the resolver to inspect the object type

To enable the type resolver, we need to add the product object to the `make_executable_schema` function under web/schema.py:

```
from pathlib import Path

from ariadne import make_executable_schema

from web.queries import query
from web.types import product_type


schema = make_executable_schema(
    (Path(__file__).parent / 'schema.graphql').read_text(),
    [query, product_type]
)
```

If we run the `allProducts` query again, we'll get a successful response. You have just learned to implement type resolvers and to handle queries that return multiple types! In the next section, we continue exploring queries by learning how to handle query parameters.

10.4.5 Handling query parameters

In this section, we learn to handle query parameters in the resolvers. Most of the queries in the products API accept filtering parameters, and all of the mutations require at least one parameter. Let's see how we access parameters by studying one example from the products API: the `products` query. The `products` query accepts an `input` filter object, whose type is `ProductsFilter`. How do we access this filter object in a resolver?

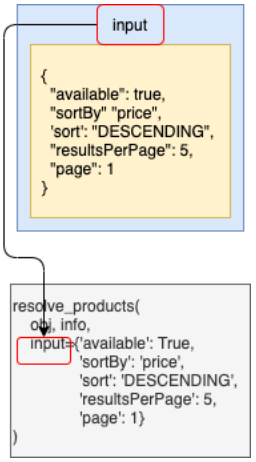


Figure 10.9 Query parameters are passed to our resolvers as keyword arguments. This example illustrates how resolve_products resolver is called, with the input parameter passed as a keyword argument. The parameter input is an object of type ProductsFilter and therefore it comes in the form of a dictionary.

As you can see in figure 10.9, when a query or mutation takes parameters, Ariadne passes those parameters to our resolvers as keyword arguments. Listing 10.8 shows how we access the input parameter for the `products` query resolver. Since the input parameter is optional and therefore nullable, we set it by default to `None`. The `input` parameter is an instance of the `ProductsFilter` input type, so when it's present in the query, it comes in the form of a dictionary. From the API specification, we know that `ProductsFilter` guarantees the presence of the following fields:

- `available`: Boolean field which filters products by whether they're available.
- `sortBy`: an enumeration type that allows us to sort products by price or name.
- `sort`: enumeration type that allows us to sort the results in ascending or descending order.
- `resultsPerPage`: indicates how many results should be shown per page.
- `page`: indicates which page of the results we should return.

In addition to these parameters, `ProductsFilter` may also include two optional parameters: `maxPrice`, which filters results by maximum price, and `minPrice`, which filters results by minimum price. Since `maxPrice` and `minPrice` are not required fields, we check for their presence using Python dictionary's get method, which returns `None` if they're not found. Let's implement the filtering and sorting functionality first, and deal with pagination afterwards. Listing 10.8 shows how we can use these parameters to filter and sort the results. The code for listing 10.8 goes under web/queries.py.

Listing 10.8 Accessing input parameters in a resolver

```
# file: web/queries.py

...

Query = QueryType()

...


@query.field('products')    #A
def resolve_products(*_, input=None):    #B
    filtered = [product for product in products]    #C
    if input is None:    #D
        return filtered
    filtered = [    #E
        product for product in filtered
        if product['available'] is input['available']
    ]
    if input.get('minPrice') is not None:    #F
        filtered = [
            product for product in filtered
            if product['price'] >= input['minPrice']
        ]
    if input.get('maxPrice') is not None:
        filtered = [
            product for product in filtered
            if product['price'] <= input['maxPrice']
        ]
    filtered.sort(    #G
        key=lambda product: product[input['sortBy']],
        reverse=input['sort'] == 'DESCENDING'
    )
    return filtered    #H
```

#A We bind a resolver for the products query by using QueryType's field decorator
#B We ignore Ariadne's resolvers default positional arguments, and instead capture the input query parameter
#C We copy the list of products so that we can make changes to the list without affecting the global in-memory list
#D If input is None, there are no filters, and therefore we return the whole dataset
#E If input is not None, we parse the filters. First, we filter products by availability
#F Then we check optional filters, starting with 'minPrice'
#G We sort the filtered result using the query's 'sort' and 'sortBy' fields
#H Finally, we return the filtered result

Let's run a query to test this resolver:

```
{
  products(input: {available: true}) {
    ...on ProdcutBase {
      name
    }
  }
}
```

You should get a valid response from the server. Now that we have filtered the results, we need to paginate them. Listing 10.9 adds a generic pagination function called `get_page` to web/queries.py. Just a word of warning: in normal circumstances, you'll be storing your data

in a database and delegating filtering and pagination to the database. The examples here are to illustrate how you use the query parameters in the resolver. We paginate the results using the `islice` function from the `itertools` module.

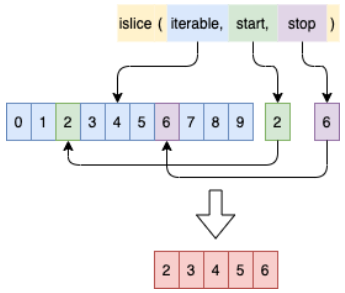


Figure 10.10 The islice function from the itertools module allows you to get a slice of an iterable object by selecting the start and stop indexes of the subset that you want to slice.

As you can see in figure 10.10, `islice` allows us to extract a slice of an iterable object. `islice` requires us to provide the start and the stop indexes of the portion that we want to slice. For example, for a list of 10 items comprising the numbers 0 to 9, providing a start index of 2 and a stop index of 6 would give us a slice with the following items: `[2, 3, 4, 5]`.

Listing 10.9 Paginating results

```
# file: web/queries.py
From itertools import islice    #A

from ariadne import QueryType

from web.data import ingredients, products


...


def get_page(items, items_per_page, page):
        start = items_per_page * page    #B
        stop = start + items_per_page    #C
        return list(islice(items, start, stop))    #D

@query.field('products')
def resolve_products(*_, input=None):
    ...
    return get_page(filtered, input['resultsPerPage'], input['page'])    #E
```

#A We import the islice function from the itertools module, which will help us to slice the list of products into pages
#B The start index is given by the page requested by the user multiplied by the number of items per page
#C We calculate the stop index by adding the number of items per page to the start index
#D We use islice to get the portion of the list that we need
#E We use the get_page function at the end of the products resolver to paginate the filtered list of products

Our hardcoded dataset only contains two products, so let's test the pagination with `resutlsPerPage` set to 1, which will split the list into two pages:

```
{
  products(input: {resultsPerPage: 1, page: 1}) {
    ...on ProdcutBase {
      name
    }
  }
}
```

You should get exactly one result. Once we implement the `addProduct` mutation in the next section, we'll be able to add more products through the API and make more use of the pagination parameters.

You've just learned how to handle query parameters! We're now in a good position to learn how to implement mutations. Mutation resolvers are similar to query resolvers, but they always have parameters. But that's enough spoiler, move on to the next section to learn more about mutations!

10.4.6 Implementing mutation resolvers

In this section, we learn to implement mutation resolvers. Implementing a mutation resolver follows the same guidelines we saw for queries. The only difference is the class we use to bind the mutation resolvers. While queries are bound to an instance of the `QueryType` class, mutations are bound to an instance of the `MutationType` class.

Let's have a look at implementing the resolver for the `addProduct` mutation. From the specification, we know that the `addProduct` mutation has three required parameters: `name`, `type`, and `input`. The shape of the `input` parameter is given by the `AddProductInput` object type. `AddProductInput` defines additional properties that can be set when creating a new product, all of which are optional and therefore nullable. Finally, the `addProduct` mutation must return a product type.

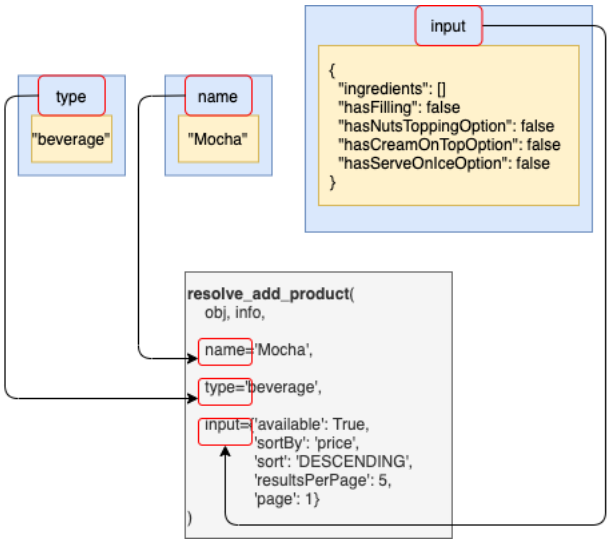


Figure 10.11 Mutation parameters are passed to our resolvers as keyword arguments. This example illustrates how the resolve_add_product resolver is called, with the name, type, and input parameters passed as keyword arguments.

Listing 10.10 shows how we implement the resolver for the `addProduct` mutation (check out figure 10.11 for an illustration). We first import the `MutationType` bindable class and instantiate it. We then declare our resolver and bind it to `MutationType` using its `field` decorator. We don't need to use Ariadne's default positional parameters `obj` and `info`, so we skip them using and wildcard followed by an underscore (`*_`), which is Python's standard syntax for naming throwaway variables. We don't set default values for `addProduct`'s parameters, since the specification states they're all required. `addProduct` must return a valid `Product` object, so we build the object with its expected attributes in the body of the resolver. Since `Product` is the union of the `Cake` and the `Beverage` types, and each type requires different sets of properties, we check the `type` parameter to determine which fields we should add to our object. The code in listing 10.10 goes into the web/mutations.py file.

Listing 10.10 Resolver for the addProduct mutation

```
# file: web/mutations.py

import uuid

from ariadne import MutationType

from web.data import products


mutation = MutationType()    #A


@mutation.field('addProduct')    #B
def resolve_add_product(*_, name, type, input):    #C
    product = {    #D
        'id': uuid.uuid4(),    #E
        'name': name,
        'available': input.get('available', False),    #F
        'ingredients': input.get('ingredients', []),
    }
    if type == 'cake':    #G
        product.update({
            'hasFilling': input['hasFilling'],
            'hasNutsToppingOption': input['hasNutsToppingOption'],
        })
    else:
        product.update({
            'hasCreamOnTopOption': input['hasCreamOnTopOption'],
            'hasServeOnIceOption': input['hasServeOnIceOption'],
        })
    products.append(product)    #H
    return product
```

#A We create a bindable for mutations by creating an instance of the MutationType class
#B We bind a resolver for the addProduct mutation by using MutationType's field decorator
#C We capture addProduct's parameters, which come in the form of keyword arguments
#D We declare the new product as a dictionary
#E The product type requires some fields that must be added in the server, such as the product ID
#F We check whether the user has set the availability of the new product in the payload, and if they haven't, we set the value by default to False
#G Beverage and Cake have different sets of required properties, so we check the type of the product to make sure we populate it with the correct set of properties
#H Finally, we return the newly created product

To enable the resolver implemented in listing 10.10, we need to add the `mutation` object to the `make_executable_schema` function in web/schema.py:

```
from pathlib import Path

from ariadne import make_executable_schema

from web.mutations import mutation
from web.queries import query
from web.types import product_type

schema = make_executable_schema(
    (Path(__file__).parent / 'schema.graphql').read_text(),
    [query, mutation, product_type]
)

server = GraphQL(schema, debug=True)
```

Let's put the new mutation at work by running a simple test. Go to the Apollo Playground running on http://127.0.0.1:8000 and run the following mutation:

```
mutation {
  addProduct(name: "Mocha", type: beverage, input:{}) {
    ...on ProductInterface {
      name,
      id
    }
  }
}
```

You'll get valid response, and a new product will be added to our list! To verify things are working correctly, run the following mutation and check that the response contains the new item just created:

```
{
  allProducts {
    ...on ProductInterface {
      name
    }
  }
}
```

Remember that we are running the service with an in-memory list representation of our data, so if you stop or reload the server, the list will be reset and you'll lose any newly created data.

You just learned how to build mutations! This is a powerful feature: with mutations, you can create and update data in a GraphQL server. We've now covered nearly all the major aspects of the implementation of a GraphQL server. In the next section, we'll take this further by learning how to implement resolvers for custom scalar types.

10.4.7 Building resolvers for custom scalar types

In this section, we learn how to implement resolvers for custom scalar types. As we saw in section 8.2.1, GraphQL provides decent amount of scalar types, such as Boolean, Integer, and String. And in many cases, GraphQL's default scalar types are sufficient to develop an API. Sometimes, however, we need to define our own custom scalars. The products API contains a custom scalar called `Datetime`. The `lastUpdated` field in both the `Ingredient` and

the `Product` types have a `Datetime` scalar type. Since `Datetime` is a custom scalar, Ariadne doesn't know how to handle it, so we need to implement a resolver for it. How do we do that?

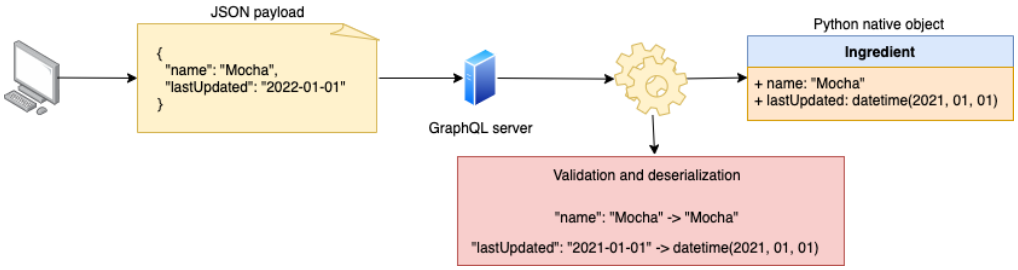


Figure 10.12 When a GraphQL server receives data from the user, it validates and deserializes the data into native Python objects. In this example, the server deserializes the name "Mocha" into a Python string, and the date "2021-01-01" into a Python datetime.

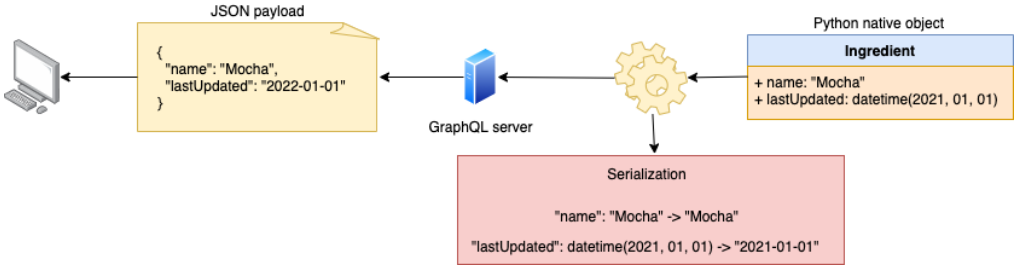


Figure 10.13 When the GraphQL server sends data to the user, it transforms native Python objects into serializable data. In this example, the server serializes the both the name and the date as strings.

As you can see in figures 10.12 and 10.13, when we encounter a custom scalar type in a GraphQL API, we need to make sure we can perform the following three actions on the custom scalar:

- **Serialization**: when a user requests data from the server, Ariadne has to be able to serialize the data (check section 2.4 for a refresher on what serialization is). Ariadne knows how to serialize GraphQL's built-in scalars, but for custom scalars, we need to implement a custom serializer. In the case of the `Datetime` scalar in the products API, we have to implement a method to serialize a datetime object.
- **Deserialization**: when a user sends data to our server, Ariadne deserializes the data and makes it available to us as a Python native data structure, such as a dictionary. If the data includes a custom scalar, we need to implement a method that lets Ariadne know how to parse and load the scalar into a native Python data structure. For the `Datetime` scalar, we want to be able to load it as a datetime object.
- **Validation**: GraphQL enforces validation of each scalar and type, and Ariadne knows how to validate GraphQL's built-in scalars. For custom scalars, we have to implement our own validation methods. In the case of the `Datetime` scalar, we want to make sure it has a valid ISO format.

Ariadne provides a simple API to handle these actions through its `ScalarType` class. The first thing we need to do is create an instance of this class:

```
from ariadne import ScalarType

datetime_scalar = ScalarType('Datetime')
```

`ScalarType` exposes decorator methods that allow us to implement serialization, deserialization, and validation. For serialization, we use `ScalarType`'s `serializer` decorator. We want to serialize datetime objects into ISO standard datetimes, and Python's datetime library provides a convenient method for ISO formatting: the `isoformat()` method:

```
@datetime_scalar.serializer
def serialize_datetime(value):
  return value.isoformat()
```

For validation and deserialization, `ScalarType` provides the `value_parser` decorator. When a user sends data to the server containing a `Datetime` scalar, we expect the date to be in ISO format and therefore parseable by Python's `datetime.fromisoformat()` method:

```
from datetime import datetime


@datetime_scalar.value_parser
def parse_datetime_value(value):
  return datetime.fromisoformat(value)
```

If the date comes in the wrong format, `fromisoformat()` will raise a `ValueError` which will be caught by Ariadne and shown to the user with the following message: "Invalid isoformat string". Listing 10.11 shows the full implementation of the `Datetime` scalar parsing and serialization. The code for listing 10.11 goes under web/types.py since it implements a type resolver.

Listing 10.11 Serializing and parsing a custom scalar

```
import uuid
from datetime import datetime

from ariadne import UnionType, ScalarType

...

datetime_scalar = ScalarType('Datetime')    #A


@datetime_scalar.serializer    #B
def serialize_datetime_scalar(date):    #C
    return date.isoformat()    #D


@datetime_scalar.value_parser    #E
def parse_datetime_scalar(date):    #F
    return datetime.fromisoformat(date)    #G
```

#A We create a bindable object for the custom Datetime scalar using the ScalarType class
#B We bind a serilzer using ScalarType's serializer decorator
#C The serializer only gets one argument, which is the object we need to serialize
#D We serialize the date object by invoking the isoformat() method
#E We bind a resolver that knows how to parse and validate a Datetime object by using ScalarType's value_parser decorator
#F The parsing function only gets one argument, which is the value we need to parse
#G To parse a date, we use datetime's fromisoformat() method

To enable the `Datetime` resolvers, we add `datetime_scalar` to the array of bindable objects for the `make_executable_schema` function under web/schema.py:

```
from pathlib import Path

from ariadne import make_executable_schema

from web.mutations import mutation
from web.queries import query
from web.types import product_type, datetime_scalar

schema = make_executable_schema(
    (Path(__file__).parent / 'schema.graphql').read_text(),
    [query, mutation, product_type, datetime_scalar]
)
```

Let's put the new resolvers to the test! Go back to the Apollo Playground running on http://127.0.0.1:8000 and execute the following query:

```
{
  allProducts {
    ...on ProductInterface {
      name,
      lastUpdated
    }
  }
}
```

You should get a list of all products with their names, and with an ISO-formatted date in the `lastUpdated` field. You now have the power to implement your own custom scalar types in GraphQL: use it wisely! Before we close the chapter, there's one more topic that we need to explore: implementing resolvers for the fields of an object type. What does that even mean? Move on to the next section to find out!

10.4.8 Implementing field resolvers

In this section, we learn to implement resolvers for the fields of an object type. We've implemented nearly all the resolvers that we need to serve all sorts of queries on the products API, but there's still one type of query that our server can't resolve: queries involving fields that map to other GraphQL types. For example, the `Products` type has a field called `ingredients`, which maps to an array of `IngredientRecipe` objects. According to the specification, the shape of the `IngredientRecipe` type looks like this:

```
type IngredientRecipe {
    ingredient: Ingredient!
    quantity: Float!
    unit: String!
}
```

Each `IngredientRecipe` object has an `ingredient` field which maps to an `Ingredient` object type. This means that, when we query the ingredients field of a product, we should be able to pull information about each ingredient, such as its name, description, or supplier information. In other words, we should be able to run the following query against the server:

```
  allProducts {
    ...on ProductInterface {
      name,
      ingredients {
        quantity,
        unit,
        ingredient{
          name
        }
      }
    }
  }
}
```

If you run this query in Apollo Playground right now, you'll get an error with the following message: "Cannot return null for non-nullable field Ingredient.name."

Why is this happening? If you look at the list of products in listing 10.5, you'll notice that the ingredients field maps to an array of objects with three fields: ingredient, quantity, and unit. For example, the Walnut Bomb has the following ingredients:

```
'ingredients': [
  {
    'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',
    'quantity': 100.00,
    'unit': 'LITRES',
  }
]
```

The `ingredient` field maps to an ingredient ID, not a full ingredient object. This is our internal representation of the product's ingredients. It's how we store product data in our database (in-memory list in this implementation). And it's a useful representation, since it allows us to identify each ingredient by ID. However, the API specification tells us that the ingredients field should map to an array of `IngredientRecipe` objects, and that each `ingredient` should represent an `Ingredient` object, not just an ID.

How do we solve this problem? We can use different approaches. For example, we could make sure that each ingredient payload is correctly built in the resolvers for each query that returns a `Product` type. For example, listing 10.12 shows how we can modify the `allProducts` resolver to accomplish this. The snippet modifies every product's `ingredients` property to make sure it contains a full ingredient payload. Since every product is represented by a dictionary, we make a deep copy of each product, to make sure the changes we apply in this function don't affect our in-memory list of products.

Listing 10.12 Update products to contain full ingredients payloads instead of just IDs

```
# file: web/queries.py

...

@query.field('allProducts')
def resolve_all_products(*_):
    products_with_ingredients = [deepcopy(product) for product in products]    #A
    for product in products_with_ingredients:
        for ingredient_recipe in product['ingredients']:
            for ingredient in ingredients:
                if ingredient['id'] == ingredient_recipe['ingredient']:
                    ingredient_recipe['ingredient'] = ingredient    #B
    return products_with_ingredients    #C
```

#A We make a deep copy of each object in the products list
#B We update the ingredient property to contain the full representation of an ingredient
#C Finally, we return the list of products with ingredients

The approach in listing 10.12 is perfectly fine, but as you can see, it makes the code grow in complexity. If we had to do this for a few more properties, the function would quickly become difficult to understand and to maintain.

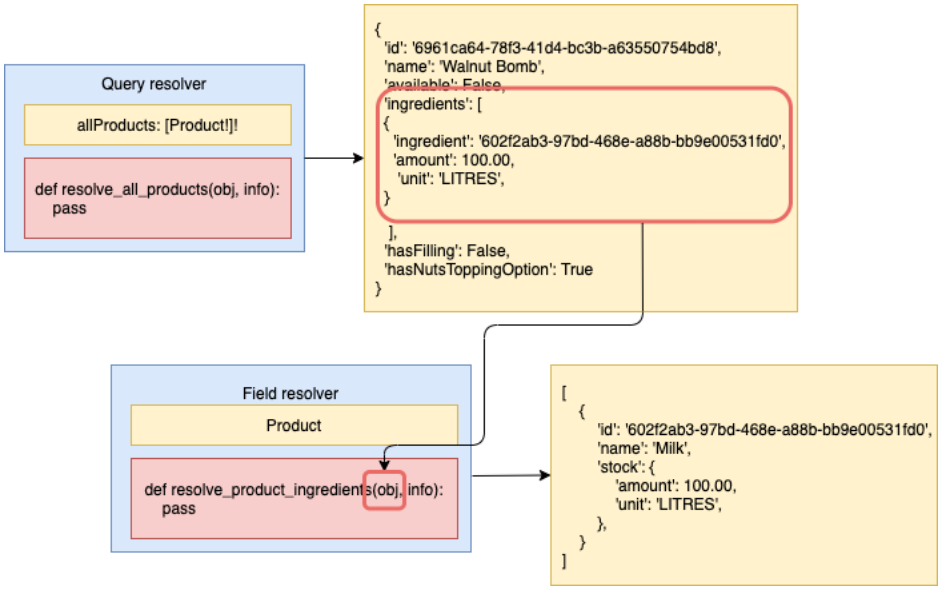


Figure 10.14 GraphQL allows us to create resolvers for specific fields of an object. In this example, the resolve_product_ingredients resolver takes care of returning a valid payload for the ingredients property of a product.

As you can see in figure 10.14, GraphQL offers an alternative way of resolving object properties. Instead of modifying the product payload within the `allProducts` resolver, we can create a specific resolver for the product's `ingredients` property, and make any necessary changes within that resolver. Listing 10.13 shows what the resolver for the product's `ingredients` property looks like. This code goes under web/types.py since it implements a resolver for object properties.

10.13 Implementing a field resolver

```
# file: web/types.py

...

@product_interface.field('ingredients')
def resolve_product_ingredients(product, _):
    recipe = [copy.copy(ingredient) for ingredient in product.get('ingredients', [])]    #A
    for ingredient_recipe in recipe:
        for ingredient in ingredients:
            if ingredient['id'] == ingredient_recipe['ingredient']:
                ingredient_recipe['ingredient'] = ingredient
    return recipe
```

#A We create a copy of each ingredient payload to make sure the changes don't affect our in-memory list of products

Object property resolvers help us to keep our code more modular because every resolver does only one thing. They also help us avoid repetition. By having a single resolver that takes care of updating the ingredients property in product payloads, we avoid having to perform this operation in every resolver that returns a product type. On the downside, property resolvers may be more difficult to trace and debug. If something is wrong with the ingredients payload, you won't find the bug within the `allProducts` resolver. You have to know that there's a resolver for products' ingredients, and look into that resolver. Application logs will help to point you in the right direction when debugging this kind of issues, but bear in mind that this design will not be entirely obvious to other developers who are not familiar with GraphQL. As with everything else in software design, make sure that code reusability doesn't impair the readability and ease of maintenance of your code.

10.5 Summary

- The Python ecosystem offers various frameworks for implementing GraphQL APIs. Check out GraphQL's official website for the latest news on available frameworks: https://graphql.org/code/.
- You can use the Ariadne framework to implement GraphQL APIs following a schema-first approach. Schema-first means we first design the API, and then we implement the server against the specification. This approach is beneficial, since it allows the server and the client development team to work in parallel.
- Ariadne can validate request and response payloads automatically using the specification, which means we don't have to spend time implementing custom validation models.
- For each query and mutation in the API specification, we need to implement a resolver. A resolver is a function that knows how to process the request for a given query or mutation. Resolvers are the code that allow us to expose the capabilities of a GraphQL API, and therefore represent the backbone of the implementation.
- To register a resolver, we use one of Ariadne's bindable classes, such as `QueryType` or `MutationType`. These classes expose decorators that allow us to bind a resolver function.
- GraphQL specifications can contain complex types, such as union types. A union type is a type which combines two or more object types. If our API specification contains a union type, we must implement a resolver that know how to determine the type of an object, since otherwise the GraphQL server doesn't know how to resolve it.
- With GraphQL, we can define custom scalars. If the specification contains a custom scalar, we must implement resolvers that know how to serialize, parse, and validate the custom scalar type, since otherwise the GraphQL server doesn't know how to handle them.

11

Authorizing access to your APIs

This chapter covers

- Using Open Authorization to allow access to our APIs
- Using OpenID Connect to verify the identity of our API users
- Which kinds of authorization flows exist, and which flow is more suitable for each authorization scenario
- Understanding JSON Web Tokens (JWT) and using Python's PyJWT library to produce and validate them
- Adding authentication and authorization to our APIs
- Integrating with an identity-as-a-service provider

In 2018, a weakness in the API authentication system of the US postal system (https://usps.com) allowed hackers to obtain data from 60 million users, including their email addresses, phone numbers, and other personal details.[1] API security attacks like this have become more and more common, with an estimated growth of over 300 percent in the number of attacks performed in 2021.[2] API vulnerabilities don't only risk exposing sensitive data from your users – they can also put you out of business![3] The good news is, there're steps you can take to reduce the risk of API breach. The first line of defense is a robust authentication and authorization system. In this chapter, you'll learn to prevent unauthorized access to your APIs by adding a robust authentication and authorization layer to your APIs.

In my experience, API authentication and authorization are two of the most confusing topics for developers, and it's also an area where implementation mistakes happen often.

[1] The issue was reported first by Brian Krebs, "USPS Site Exposed Data on 60 Million Users", KrebsOnSecurity, Nov 21st 2018 (https://krebsonsecurity.com/2018/11/usps-site-exposed-data-on-60-million-users/ [accessed 4th Jan 2022]).

[2] Bill Doerfeld, "API Attack Traffic Grew 300+% In the Last Six Months", *Security Boulevard*, July 30th 2021 (https://securityboulevard.com/2021/07/api-attack-traffic-grew-300-in-the-last-six-months/ [accessed 4th Jan 2022]).

[3] Galvin, Joe, "60 Percent of Small Businesses Fold Within 6 Months of a Cyber Attack", *Inc.*, May 7 2018 (https://www.inc.com/joe-galvin/60-percent-of-small-businesses-fold-within-6-months-of-a-cyber-attack-heres-how-to-protect-yourself.html [accessed 4th Jan 2022]).

Before you go ahead and implement the security layer of your API, I highly recommend you read this chapter to make sure you know what you're up to, and how to do it correctly. I've done my best to provide a comprehensive summary of how API authentication and authorization works, and by the end of this chapter you should be able to add a robust authentication flow to your own APIs.

Authentication is the process of verifying the identity of a user, while authorization is the process of determining whether a user has access to certain resources or operations. The concepts and standards about authentication and authorization that you'll learn in this chapter are applicable to all types of web APIs.

You'll learn different authentication and authorization protocols and flows, and how to validate authentication tokens. You'll also learn to use Python's PyJWT library to produce signed tokens and to validate them. We'll walk through a practical example of adding authentication and authorization to the orders API.

We can use various strategies to handle authentication and authorization. You can build your own authentication service, or you can use an identity-as-a-service provider, such as Auth0, Okta, Azure Active Directory, or AWS Cognito. Unless you're an expert in web security and authentication protocols and have sufficient resources to build the system correctly, I recommend you use an identity service provider. In this chapter, we'll learn to add authentication to our APIs with Auth0, which is one of the most popular identity management systems.

We'll use Auth0's free plan. Auth0 takes care of managing user accounts, issuing secure tokens, and it also provides easy integrations for social login with identity providers such as Google, Facebook, Twitter, and others. Auth0's authentication system is built on standards, so everything you learn about authenticating with Auth0 applies to any other provider. If you use a different authentication system in your own projects or at work, you'll be able to take the lessons from this chapter and apply them to whichever other system you use.

We've got a lot to cover, so let's get started!

11.1 Setting up the environment for this chapter

Before we get started, let's set up the environment for this chapter. The code for this chapter is available under the directory called ch11 in the GitHub repository for this book. Under ch11, you'll find two folders: ch11/orders and ch11/ui. I'll explain how to set up the UI in section 11.7. This section explains how to set up the code for the orders API, which lives under the ch11/orders folder.

In chapter 7, we implemented a fully functional orders service, complete with its business layer, database, and its API. This chapter picks up the orders service from where we left it in chapter 7. So, if you want to follow along with the changes in this chapter, create a folder called ch11, `cd` into it, and copy over the implementation from chapter 7 with the following command:

```
$ cp -r ../ch07 orders
```

Now `cd` into orders and install the dependencies by running `pipenv install`. For this chapter, we need a few additional dependencies, so run the following command to install them:

```
$ pipenv install cryptography pyjwt
```

PyJWT is a Python library that allows us to work with JSON Web Tokens, while cryptography will allow us to verify the tokens' signatures.

Our environment is now ready, so let's begin our journey through the wondrous world of user authentication and authorization. It's a journey full of pitfalls, yet a necessary one. So, hold tight, and watch carefully as we go along!

11.2 Understanding authentication and authorization protocols

When it comes to API authentication, the two most important protocols you need to know are OAuth (Open Authorization) and OpenID Connect. This section explains how each protocol works and how they fit within the authentication and authorization flows for our APIs.

11.2.1 Understanding Open Authorization

Open Authorization is a standard protocol for access delegation[4]. As you can see in figure 11.1, OAuth allows an application user to grant a third-party application access to protected resources they own in another website, without having to share their credentials.

> **DEFINITION: OPEN AUTHORIZATION (OAUTH)** is an open standard that allows users to grant access to third-party applications to their information on other websites. Typically, access is granted by issuing a token which the third-party application uses to access the user's information.

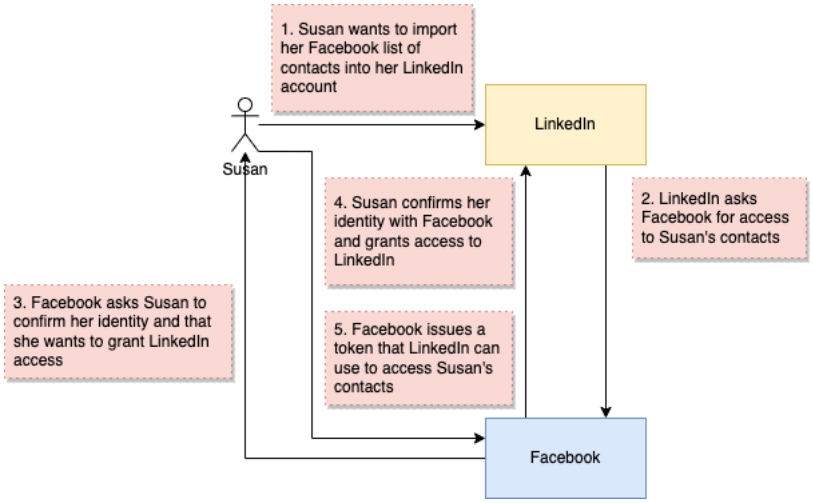


Figure 11.1 With Open Authorization, a user can grant access to a third-party application to their information in another website.

[4] https://oauth.net/ is a pretty good website with tons of resources to learn more about OAuth and check the specifications.

For example, let's say Susan has a list of contacts in her Facebook account. One day, Susan signs into LinkedIn, and she wants to import her list of contacts from Facebook. To allow LinkedIn to import her Facebook contacts, Susan has to grant LinkedIn access to that resource. How can she grant LinkedIn access to her list of contacts? She could give LinkedIn her Facebook credentials to access her account. But that would be a major security vulnerability. Instead, OAuth defines a protocol which allows Susan to tell Facebook that LinkedIn can access her list of contacts. With OAuth, Facebook issues a temporary token which LinkedIn can use to import Susan's contacts.

OAuth distinguishes various roles in the process of granting access to a resource:

- **Resource owner**: the user who's granting access to the resource. In the previous example, Susan is the resource owner.
- **Resource server**: the server hosting the user's protected resources. In the previous example, Facebook is the resource server.
- **Client**: the application requesting access to the user's resources. In the previous example, LinkedIn is the client.
- **Authorization server**: the server which grants the client access to the resources. In the previous example, Facebook is the authorization server.

OAuth offers four different flows to grant authorization to a user depending on the access conditions. It's important to know how each flow works and in which scenarios you can use it in. In my experience, OAuth flows are one of the biggest areas of confusion around authorization, and one of the biggest sources of security problems in modern websites. These are the OAuth flows:

- Authorization code flow
- Implicit flow
- Resource owner password flow
- Client credentials flow

Let's delve into each flow to understand how they work and when we use them!

AUTHORIZATION CODE FLOW

In the authorization code flow, the client server exchanges a secret with the authorization server to produce a signing URL. As you can see in figure 11.2, after the user signs in this URL, the client server obtains a one-time code which it can exchange for an access token. This flow uses a client secret and therefore it's only appropriate for applications in which the code is not publicly exposed, such as traditional web applications where the user interface is rendered in the backend.

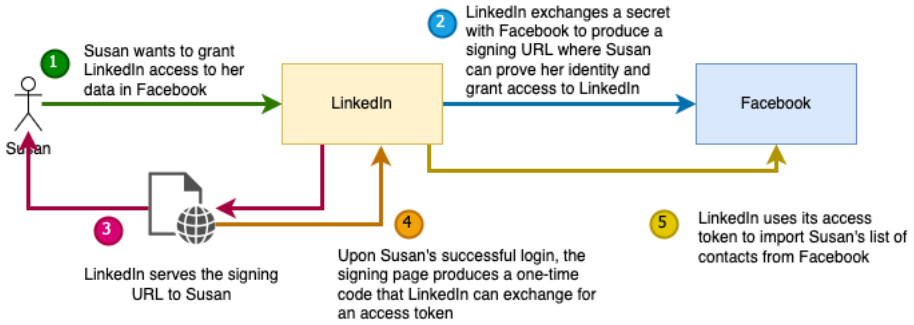


Figure 11.2 In the authorization code flow, the authorization server produces a signing URL with which the user can prove their identity and grant access to the third-party application.

IMPLICIT FLOW

As illustrated in figure 11.3, in the implicit fow the client gets the access token directly from the authentication service. This flow is suitable for applications with a publicly exposed source code, such as Single Page Applications (SPAs). SPAs are JavaScript applications that run in the browser and their source code is accessible to everybody, so the implicit flow code is the right choice for them.

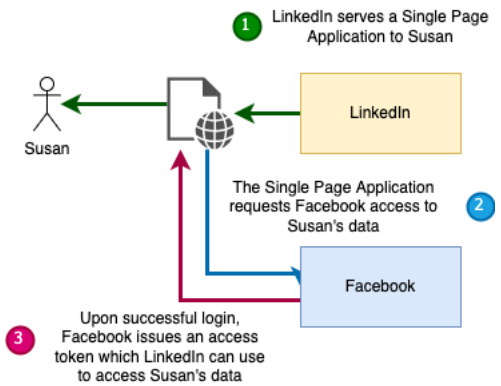


Figure 11.3 In the implicit flow, a Single Page Application served by the client requests access to the user's data directly from the authorization server.

RESOURCE OWNER PASSWORD FLOW

As you can see in figure 11.4, in the resource owner password flow, the user sends his credentials directly to the client server (LinkedIn), which then exchanges them with the authorization server (Facebook) for an access token. Since user credentials are directly exposed to the application server, this flow is generally not recommended and should only be used with highly trusted application servers.

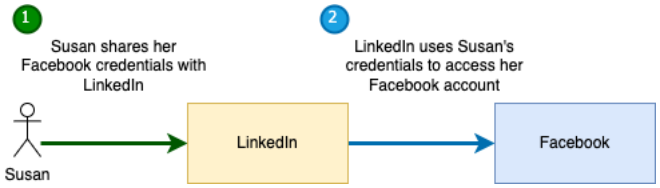


Figure 11.4 In the resource owner password flow, the user shares their credentials with the third-party application to grant it access to their data in a different website.

CLIENT CREDENTIALS FLOW

The client credentils flow is aimed for server-to-server communication, and as you can see in figure 11.5, it involves the exchange of a secret to obtain an access token. This flow is suitable for enabling communication between microservices. We'll see an example of this flow in section 11.7.

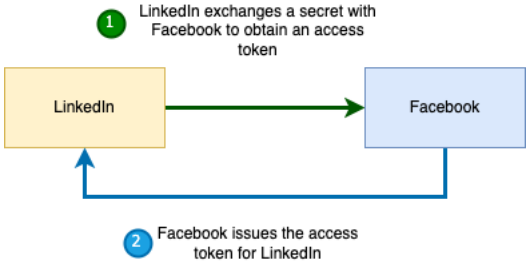


Figure 11.5 In the client credentials flow, a server application exchanges a secret with the authorization server to obtain an access token.

Now that we understand how Open Authorization works, let's turn our attention to OpenID Connect!

11.2.2 Understanding OpenID Connect

OpenID Connect (OIDC) is an open standard for identify verification that's built on top of Open Authorization. As you can see in figure 11.6, OIDC allows users to authenticate to a website by using a third-party identity provider. If you've used your Facebook, Twitter, or your Google account to sign into other websites, you're already familiar with OIDC. In this case, Facebook, Twitter, and Google are identity providers. You use them to bring your identity to a new website. OIDC is a convenient authentication system, since it allows users to use the same identity across different websites, without having to create and manage new usernames and passwords.

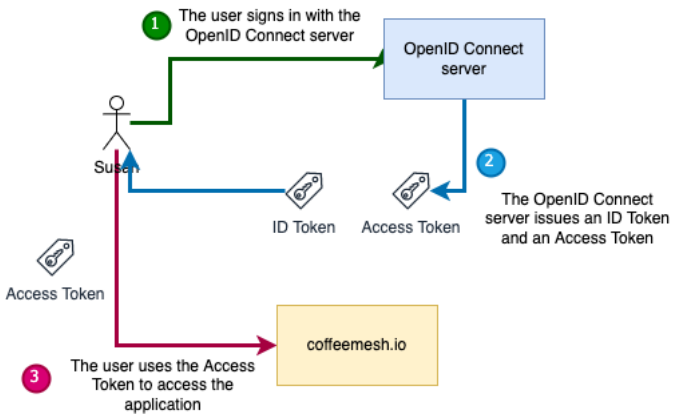


Figure 11.6 With OpenID Connect (OIDC), a user signs-in with an OIDC server. The OIDC server issues an ID Token and an Access Token which the user can use to access an application.

> **DEFINITIONS: OPENID CONNECT (OIDC)** is an identity verification protocol that allows users to bring their identity from one website (the identity provider) to another. OIDC is built on top of OAuth and we can use the same flows defined by OAuth to authenticate users.

Since OIDC is built on top of Open Authorization, we can use any of the authorization flows described in the previous section to authenticate and authorize our users. As you can see in figure 11.6, when we authenticate using the OIDC protocol, we distinguish two types of tokens: **ID tokens** and **access tokens**. Both tokens come in the form of JSON Web Tokens, but they serve different purposes: **ID tokens** identify the user, and they contain information such as the user's name, their email, and other personal details. You must use ID tokens only to verify the user identity, and never to determine whether a user has access to an API. API access is validated with access tokens. **Access tokens** typically don't contain user information, and instead contain a set of claims about the access rights of the user.

> **ID TOKENS VS ACCESS TOKENS** A common security problem is the misuse of ID tokens and access tokens. ID tokens are tokens which carry the identity of the user. The must be used exclusively for verifying the user's identity, and not for validating access to an API. API access is validated through access tokens. Access tokens rarely contain identity details of the user, and instead they contain claims about the user's right to access the API. A fundamental difference between ID tokens and access tokens is the audience: the ID token's audience is the authorization server, while the access token's audience is our API server.

Identity providers that offer OIDC integrations expose a /.well-known/openid-configuration endpoint (with a leading period!), also known as the discovery endpoint. This endpoint tells the API consumer how to authenticate and obtain their access tokens. For example, the OIDC's well-known endpoint for Google Accounts is https://accounts.google.com/.well-known/openid-configuration. If you call this endpoint, you'll obtain the following payload (the example is truncated with an ellipsis):

```
{
  "issuer": "https://accounts.google.com",
  "authorization_endpoint": "https://accounts.google.com/o/oauth2/v2/auth",
  "device_authorization_endpoint": "https://oauth2.googleapis.com/device/code",
  "token_endpoint": "https://oauth2.googleapis.com/token",
  "userinfo_endpoint": "https://openidconnect.googleapis.com/v1/userinfo",
  "revocation_endpoint": "https://oauth2.googleapis.com/revoke",
  "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
  "response_types_supported": [
    "code",
    "token",
    "id_token",
    "code token",
    "code id_token",
    "token id_token",
    "code token id_token",
    "none"
  ],
  ...
}
```

As you can see, the well-known endpoint tells us which URL we must use to obtain the authorization access token, which URL returns user information, or which URL we use to revoke an access token. There're other bits of information in this payload, such as available claims or the JSON Web Keys (JWKS) endpoint, whose meaning will become clear later in this chapter. Typically, you'll use a library to handle these endpoints on your behalf, or you'll use an identity-as-a service provider to take care of these integrations. If you want to learn more about OpenID Connect, I recommend Prabath Siriwardena's *OpenID Connect in Action* (Manning, 2022).

Now that we know how Open Authentication and OpenID Connect work, it's time get down to the details of how authentication and authorization work. We'll start by studying what JSON Web Tokens are in the next section.

11.3 Working with JSON Web Tokens

In OAuth and OpenID Connect, user access is verified by means of a token known as JSON Web Token or JWT. This section explains what JSON Web Tokens are, how they're structured, what kinds of claims they contain, and how to produce and validate them.

A JSON Web Token (JWT) is a token which represents a JSON document. The JSON document contains claims, such as who issued the token, what the audience of the token is, or when the token expires. The JSON document is typically encoded as a Base64 string. JWTs are normally signed with a private secret or a cryptographic key[5]. A typical JSON Web Token looks like this:

[5] The full specification for how JSON Web Tokens should be produced and validated is available under J. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)", RFC-7519, May 2015 (https://datatracker.ietf.org/doc/html/rfc7519).

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbWVzaC5pby8iLCJzdW
        IiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiLCJhdWQiOiJodHRwOi8vMTI3LjAuMC
        4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MjI4NDg2LjE1OTg4MSwiZXhwIjoxNjM4MzE0ODg2LjE1OTg4MS
        wic2NvcGUiOiJvcGVuaWQifQ.oblJ5wV9GqrhIDzNSzcClrpEQTMK8hZGzn1S707tDtQE__OCDsP9J2Wa70a
        Bua6X81-zrvWBfzrcX--nSyT-
        A9uQxL5j3RHHycToqSVi87I9H6jgP4FEKH6ClwZfabVwzNIy52Zs7zRdcSI4WRz1OpHoCM-
        2hNtZ67dMJQgBVIlrXcwKAeKQWP8SxSDgFbwnyRTZJt6zijRnCJQqV4KrK_M4pv2UQYqf9tQpj2uflTsVcZq
        6XsrFLAgqvAg-YsIarYw9d63rs4H_I2aB3_T_1dGPY6ic2R8WDT1_AXzi-crjoWq9A51SN-
        kMaTLhE_v2MSBB3A0zrjbdC4ZvuszAqQ
```

If you look closely at the above example, you'll see the string contains two periods. The periods act as delimiters which separate each component of the JSON Web Token. As you can see in figure 11.7, a JSON Web Token document has three sections:

- **Header**: identifies the type of token, as well as the algorithm and the key that were used sign the token. We use this information to apply the right algorithm to verify the token's signature.
- **Payload**: contains the document's set of claims. The JWT specification includes a list of reserved claims which identify the issuer of the token (the authorization server), the token's audience or intended recipient (our API server), and its expiry date, among other details. In addition to JWT's standard claims, a payload can also include custom claims. We use this information to determine whether the user has access to the API.
- **Signature**: a string representing the token's signature.

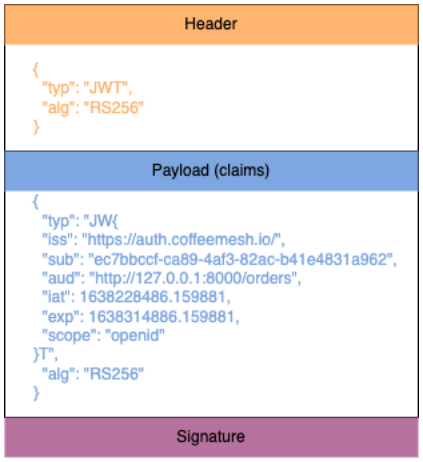


Figure 11.7 A JSON Web Token (JWT) is composed of three parts: 1) a header which contains information about the token itself; 2) a payload with claims about the user's access to the website; and 3) a signature that proves the authenticity of the token.

Now that we understand what a JSON Web Token is and what its structure looks like, let's delve deeper into its properties. The next sections explain the main types of claims and properties we can find in JWT payloads and headers and how we use them.

11.3.1 Understanding the JWT header

JSON Web Tokens contain a header which describes the type of token, as well as the algorithm and the key that that were used to sign the token. JWTs are commonly signed using the HS256 and the RS256 algorithms. HS256 uses a secret to encrypt the token, while RS256 uses a private/public key to sign the token. We use this information to apply the right algorithm to verify the token's signature.

Signing algorithms for JSON Web Tokens

The two most common algorithms used for signing JSON Web Tokens are HS256 and RS256. HS256 stands for HMAC-SHA256 and it's a form of encryption that uses a key to produce a hash. You can learn more about HMAC in David Wong's *Real-World Cryptography* (Manning, 2021) under this link: https://livebook.manning.com/book/real-world-cryptography/chapter-3/point-13852-83-93-1

RS256 stands for RSA-SHA256. RSA (Rivest-Shamir-Adleman) is a form of encryption that uses a private key to encrypt the payload. In this case, we can verify that the token's signature is correct by using a public key. To learn more about this form of encryption, head over to David Wong's *Real-World Cryptography* (Manning, 2021) [https://livebook.manning.com/book/real-world-cryptography/chapter-6/point-13853-62-153-1].

A typical JWT header is the following:

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "ZweIFRR4l1dJlVPHOoZqf"
}
```

Let's analyze this header:

- **`alg`**: tells us that the token was signed using the RS256 algorithm.
- **`typ`**: tells us that this is a JWT token.
- **`kid`**: tell us that the key used to sign the token has the ID `ZweIFRR4l1dJlVPHOoZqf`.

A token's signature can only be verified using the same secret or key that was used to sign it. For security, we often use a collection of secrets or keys to sign the tokens. The `kid` field tells us which secret or key to use to sign the token, so that we can use the right value when verifying the token's signature.

Some tokens also contain a `nonce` field in the header. If you see one of those tokens, chances are the token isn't for your API server, unless you're the creator of the token and you know what the value for `nonce` is. The `nonce` field typically contains an encrypted secret that adds an additional layer of security to the JWT. For example, the tokens issued by the Azure Active Directory to access its Graph API contain a `nonce` token, which means you shouldn't use those tokens to authorize access to your custom APIs. Now that understand the properties of a token's header, the next section explains how to read the token's claims.

11.3.2 Understanding JWT claims

The payload of a JSON Web Token contains a set of claims. Since a JSON Web Token payload is a JSON document, the claims come in the form of key-value pairs.

There are two types of claims: reserved claims, which are part of the JWT specification, and custom claims, which are claims we can add to enrich the tokens with additional information[6]. The JWT specification defines seven reserved claims:

- **`iss` (issuer)**: identifies the issuer of the JWT. If you use an identity as a service provider, the issuer identifies that service. It typically comes in the form of an ID of a URL.
- **`sub` (subject)**: identifies the subject of the JWT, i.e. the user sending the request to the server. It typically comes in the form of an opaque ID, i.e. an ID that doesn't disclose personal details of the user.
- **`aud` (audience)**: indicates the recipient for which the JWT is intended. This is our API server. It typically comes in the form of an ID or a URL. It's crucial to check this field to validate that the token is intended for our APIs. If we don't recognize the value in this field, it means the token isn't for us and we must disregard the request.
- **`exp` (expiration time)**: when the JWT expires. Requests with expired tokens must be rejected.
- **`nbf` (not before time)**: time before which the JWT must not be accepted.
- **`iat` (issued at time)**: when the JWT was issued. It can be used to determine the age of the JWT.
- **`jti` (JWT ID)**: a unique identifier for the JWT.

The reserved claims are not required in the JWT payload, but it's recommended to include them to ensure interoperability with third-party integrations. Listing 11.1 shows an example of JWT payload.

Listing 11.1 Example of JWT payload claims

```
{
  "iss": "https://auth.coffeemesh.io/",
  "sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",
  "aud": "http://127.0.0.1:8000/orders",
  "iat": 1667155816,
  "exp": 1667238616,
  "azp": "7c2773a4-3943-4711-8997-70570d9b099c",
  "scope": "openid"
}
```

Let's dissect the claims in listing 11.1:

[6] You can check a full list of the most commonly used JWT claims under this link: https://www.iana.org/assignments/jwt/jwt.xhtml.

- `iss` tells us that the token has been issued by the https://auth.coffeemesh.io server identity service.
- `sub` tell us that the user has the identifier ec7bbccf-ca89-4af3-82ac-b41e4831a962. The value of this identifier is owned by the identity service. Our APIs can use this value to control access to the resources owned by this user in an opaque way. We say this ID is opaque because it doesn't disclose any personal information about the user.
- `aud` tells us that this token has been issued to grant access to the orders API. If the value of this field is a different URL, the orders will reject the request.
- `iat` tells us that the token was issued on the 30th of October of 2021 at 6:50pm.
- `exp` tells us that the token expires on the 31st of October of 2021 at 5:50pm.
- `azp` tells us that the token has been requested by an application with identifier 7c2773a4-3943-4711-8997-70570d9b099c. This is typically a frontend application. This claim is common in tokens that have been issued using the OpenID Connect protocol.
- The `scope` field tells us that this token was issued using the OpenID Connect protocol.

Now that we know how to work with token claims, let's see how we produce and validate tokens!

11.3.3 Producing JSON Web Tokens

To form the final JSON Web Token, we encode the header, the payload, and the signature using base64url encoding. As documented in RFC 4648 (https://datatracker.ietf.org/doc/html/rfc4648), base64url encoding is similar to base64, but it uses non-alphanumeric characters and omits padding. The header, payload, and signature are then concatenated using periods as separators. Libraries like PyJWT take care of the heavy lifting of producing a JSON Web Token. Let's say we want to produce a token for the payload we saw in listing 11.1:

```
payload = {
  "iss": "https://auth.coffeemesh.io/",
  "sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",
  "aud": "http://127.0.0.1:8000/orders",
  "iat": 1667155816,
  "exp": 1667238616,
  "azp": "7c2773a4-3943-4711-8997-70570d9b099c",
  "scope": "openid"
}
```

To produce a signed token with this payload, we use PyJWT's `encode()` function, passing in the token, the key to sign the token, and the algorithm we want to use to sign the token:

```
>>> import jwt
>>> jwt.encode(payload=payload, key='secret', algorithm='HS256')
'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbWVzaC5pby8iLCJzd
       WIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiLCJhdWQiOiJodHRwOi8vMTI3LjAuM
       C4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjY3MTU1ODE2LCJleHAiOjE2NjcyMzg2MTYsImF6cCI6IjdjMjc3M
       2E0LTM5NDMtNDcxMS04OTk3LTcwNTcwZDliMDk5YyIsInNjb3BlIjoib3BlbmlkIn0.sZEXZVitCv0iVrbxG
       N54GJr8QecZfHA_pdvfEMzT1dI'
```

In this case, we're signing the token with a secret keyword using the HS256 algorithm. For a more secure encryption, we use a private/public key pair to sign the token with the RS256 algorithm. To sign JWTs, we typically use keys that follow the X.509 standard, which allows us to bind an identity to a public key. To generate a private/public key pair, run the following command from your terminal:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout private_key.pem -out public_key.pem -
      subj "/CN=coffeemesh"
```

The minimum input for a X.509 certificate is the subject's common name (CN), which in this case we set `coffeemesh`. If you omit the -subj flag, you'll be prompted with a series of questions about the identity you want to bind the certificate to. This command produces a private key under a file named ch11/private_key.pem, and the corresponding public key under a file named ch11/private_key.pem. If you're unable to run these commands, you can find a sample key pair in the repository provided with this book, under ch11/private_key.pem and ch11/public_key.pem.

Now that we have a private/public key pair, we can use them to sign our tokens and to validate them. Go ahead and create a file named ch11/jwt_generator.py and copy into it the contents of listing 11.2 shows how to generate JWT tokens signed with a private key. The listing defines a function, `generate_jwt()`, which generates a JWT for the payload defined within the function. In the payload, we set the `iat` and the `exp` properties dynamically: `iat` is set to the current UTC time, `exp` is set to 24 hours from now. We load the private key using cryptography's `serialization()` function passing in as parameters the text of our private key encoded in bytes, as well as the passphrase encoded in bytes. Finally, we encode the payload using PyJWT's `encode()` function passing in the payload, the loaded private key, and the algorithm we want to use to sign the key (RS256).

Listing 11.2 Generating JSON Web Tokens signed with a private key

```
from datetime import datetime, timedelta
from pathlib import Path

import jwt
from cryptography.hazmat.primitives import serialization


def generate_jwt():
    now = datetime.utcnow()
    payload = {
        "iss": "https://auth.coffeemesh.io/",
        "sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",
        "aud": "http://127.0.0.1:8000/orders",
        "iat": now.timestamp(),
        "exp": (now + timedelta(hours=24)).timestamp(),
        "scope": "openid",
    }

    private_key_text = Path("signing_key").read_text()
    private_key = serialization.load_pem_private_key(
        private_key_text.encode(),
        password=None,
    )
    return jwt.encode(payload=payload, key=private_key, algorithm="RS256")


print(generate_jwt())
```

To see this code at work, activate your virtual environment by running `pipenv shell` and execute the following command:

```
$ python jwt_generator.py
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbWVzaC5pby8iLCJzdW
       IiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiLCJhdWQiOiJodHRwOi8vMTI3LjAuMC
       4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MDMxLjgzOTY5ODczOTEsImV4cCI6MTYzODExOC4yMzk2OTg5OT
       MsInNjb3BlIjoib3BlbmlkIn0.GIpMvEvZG8ERmMA99geYUq5IKeWpRrnHoViLb1CkRufqC5vgM9555re4Is
       LLa7yVxNAXIpFVFBqaoWrloJl6dSQ5r00dvUBSM1EM78KMZ7f0gQqUDFWNoKWCeyQu1QCBzuHTouS4l_mzzI
       i75SAl3DJLTaj4zr6c_bQdUuDU1GyrIOJiPSCHSlnKPgg9tjrX8eOcB_ESGSo9ipnCbPAluWp0cDjPRPBNRu
       iU53sbli-
       dTy7WoCD1mXAbqhztwO39kG3DZBkysB4vTnKU4Eul2yNNYK2hHVZQEvAqq8TJjETUS7iekf0NSt1qQArJ7cx
       g6Jh5D7y5pbKmYYsBlFohPg
```

Now you know how to generate JSON Web Tokens! The JWT generator from listing 11.2 is handy for running tests, and we'll use it in the upcoming sections to test our code before we integrate with Auth0. Now that we understand how JSON Web Tokens are generated, let's see how to inspect their payloads and how to validate them.

11.3.4 Inspecting JSON Web Tokens

Often when working with JWTs you'll run into validation issues. To understand why a token validation is failing, it's useful to inspect the payload and verify whether its claims are correct. In this section, you'll learn to inspect JWTs using three different tools: https://jwt.io, the terminal's `base64` command, and with Python. To try out these tools, run the jwt_generator.py script we created in section 11.3.3 to issue a new token.

https://jwt.io is an excellent tool which offers an easy way to inspect a JWT. As you can see in figure 11.8, all you need to do is paste the JWT in the input panel on the left. The display panel on the right will show you the contents of the token's header and payload. You can also verify the token's signature by providing your public key.

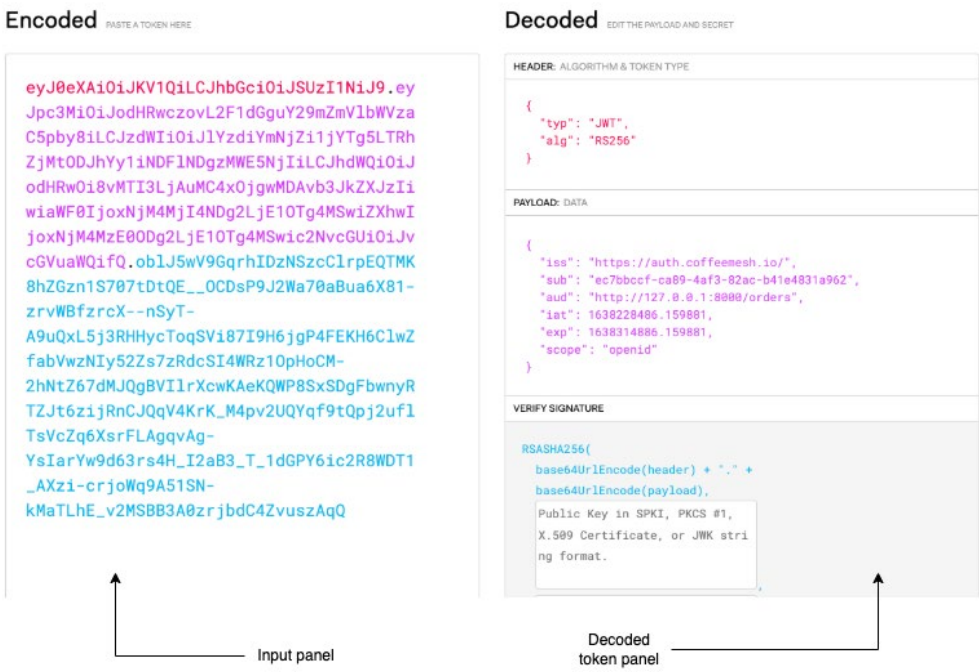


Figure 11.8 https://jwt.io is a tool that helps to easily inspect and visualize JSON Web Tokens. Simply paste the token on the left-side panel. You can also verify the token's signature by pasting the public certificate on the "VERIFY SIGNATURE" box on the right.

You can also inspect the contents of the JSON Web Token by decoding the header and payload in the terminal using the `base64` command. For example, to decode the token's header in the terminal, run the following command:

```
$ echo eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9 | base64 –decode
{"alg":"RS256","typ":"JWT"}
```

We can also inspect the contents of a JWT using Python's base64 library. To decode a JWT header with Python, open a Python shell and run the following code:

```
>>> import base64
>>> base64.decodebytes('eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9'.encode())
b'{"alg":"RS256","typ":"JWT",}'
```

Since the JWT payload is also base64url encoded, we use the same methods for decoding it. Now that we know how to inspect JWT payloads, let's see how we validate them!

11.3.5 Validating JSON Web Tokens

There're two parts to validating a JSON Web Token. On one hand, you must validate the JSON Web Token's signature, and on the other hand, you must validate that its claims are correct, for example by ensuring that the token isn't expired and that the audience is correct. This process must be crystal clear: both steps of the validation process are required. An expired token with a valid signature shouldn't be accepted by the API server, while an active token with an invalid signature isn't any good either. Every user request to the server must carry a token, and the token must be validated on each request.

> **VALIDATE JWTS ON EACH REQUEST.** When a user interacts with our API server, they must send an JSON Web Token in each request, and we must validate the token on each request. Some implementations, especially those that use the authorization code flow we discussed in section 11.2.1, store tokens in a session cache and check the request's token against the cache. That's not how JWTs are meant to be used. JWTs are designed for stateless communication between the client and the server, and therefore must be validated using the methods we describe in this section.

As we saw in section 11.3.3, tokens can be signed with a secret key or with a private/public key pair. For security, most websites use tokens that are signed with private/public keys, and to validate the signature of such tokens, we use the public key.

Let's see how we validate a token in code. We'll use the signing key we created in section 11.3.3 to produce and validate the token. Activate your pipenv environment by running `pipenv shell` and execute the jwt_generator.py script to issue a new token.

To validate the token, we must first load the public key using the following code:

```
>>> from cryptography.x509 import load_pem_x509_certificate
>>> from pathlib import Path
>>> public_key_text = Path('public_key.pem').read_text()
>>> public_key = load_pem_x509_certificate(public_key_text.encode('utf-8')).public_key()
```

Now that we have the public key available, we can use it to validate a token with the following code:

```
>>> import jwt
>>> access_token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ..."
>>> jwt.decode(token, key=public_key, algorithms=['RS256'],
       audience=["http://127.0.0.1:8000/orders"])
{'iss': 'https://auth.coffeemesh.io/', 'sub': 'ec7bbccf-ca89-4af3-82ac-b41e4831a962',
       'aud': 'http://127.0.0.1:8000/orders', 'iat': 1638114196.49375, 'exp':
       1638200596.49375, 'scope': 'openid'}
```

As you can see, if the token is valid, we'll get back the JWT payload. If the token is invalid, this code will raise an exception. Now that we know how to work with and validate JSON Web Tokens, let's see how we set up an identity-as-a-service provider for our APIs.

11.4 Adding authorization to the API server

Now that we know how to validate access tokens, let's put all this code together in our API server. In this section, we add authorization to the orders API. Some endpoints of the orders API are protected, while others must be accessible to everyone. Our goal is to ensure that our server checks for valid access tokens under the protected endpoints.

We'll allow public access to the /docs/orders and the /openapi/orders.json endpoints, since they serve the API documentation and that must be available for all consumers. All other endpoints require valid tokens. If the token is invalid or is missing in the request, we must reject the request with a 401 (Unauthorized) status code, which indicates that credentials are missing.

How do we add authorization to our APIs? There're two major strategies: 1) handling validation in an API Gateway, or 2) handling validation in each service. An API Gateway is a network layer that sits in front of our APIs[7]. The main role of an API Gateway is to facilitate service discovery, but they can also be used to authorize user access, validate access tokens, and to enrich the request with custom headers adding information about the user.

The second method is to handle authorization within each API. You'll handle authorization at the service level when your API Gateway can't handle authorization or when an API Gateway doesn't fit in your architecture. In this section, we'll learn to handle authorization within the service using this approach since we don't have an API Gateway.

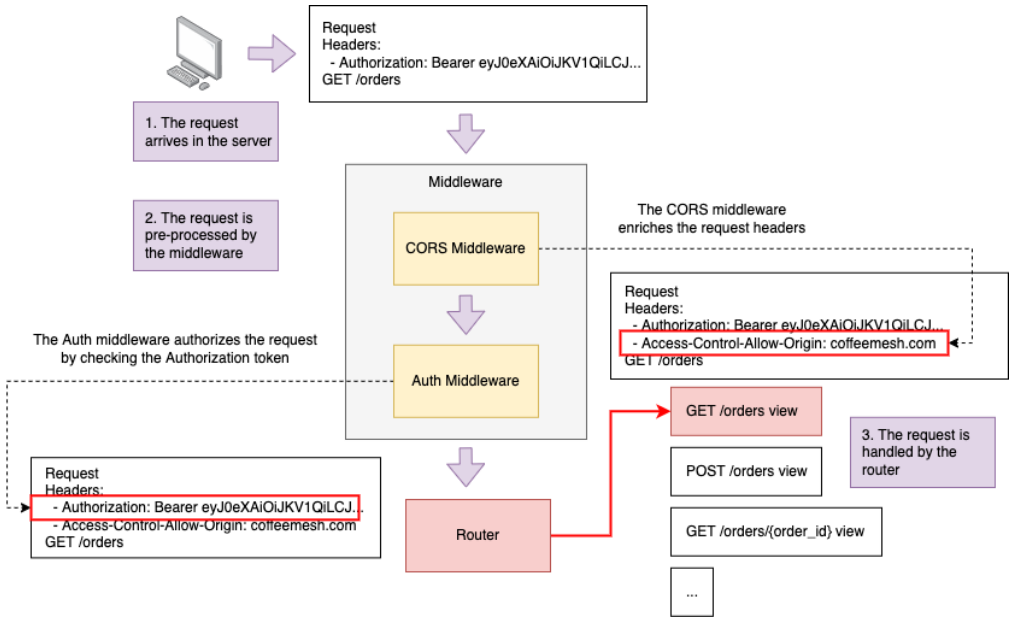


Figure 11.9 A request is first processed by the server middleware, such as the CORS and the auth middlewre, before making it to the router which maps the request to the corresponding view function.

[7] See Chris Richardson, "Pattern: API Gateway / Backends for Frontends", at https://microservices.io [accessed 8th Nov 2021].

A question that often comes up is where exactly in our code do we handle authorization? Since authorization is needed to validate user access to the service through the API, we implement in the API middleware. As you can see in figure 11.9, **middleware** is a layer of code that provides common functionality to process all our requests. Most web servers have a concept of middleware or request pre-processors, and that's where our authorization code goes. Middleware components usually are executed in order, and typically we can choose the order in which they're executed. Since authorization controls access to our server, the authorization middleware must be executed early.

11.4.1 Creating an authorization module

Let's first create a module to encapsulate our authorization code. Create a file named ch11/orders/web/api/auth.py and copy the code in listing 11.3 into it. We start by loading the public key we created in section 11.3.3. To validate the token, we first retrieve the headers and load the public key. We use PyJWT's `decode()` function to validate the token, passing in as parameters the token itself, the public key required to validate the token, the expected list of audiences, and the algorithms used to sign the key.

Listing 11.3 Adding an authorization module to the API

```
from pathlib import Path

import jwt
from cryptography.x509 import load_pem_x509_certificate


public_key_text = (Path(__file__).parent / "../../../public_key.pem").read_text()
public_key = load_pem_x509_certificate(public_key_text.encode()).public_key()


def decode_and_validate_token(access_token):
    """
    Validates an access token. If the token is valid, it returns the token payload.
    """
    return jwt.decode(
        access_token,
        key=public_key,
        algorithms=["RS256"],
        audience=["http://127.0.0.1:8000/orders"],
    )
```

Now that we created a module that encapsulates the functionality necessary to validate an JWT, let's incorporate it into the API by adding a middleware that uses it to validate access to the API.

11.4.2 Creating an authorization middleware

To add authorization to our API, we create an authorization middleware. Listing 11.4 shows how to implement the authorization middleware. The code in listing 11.4 goes into the ch11/orders/app.py file, with the newly added code in bold case. We implement the middleware as a simple class called `AuthenticateRequestMiddleware`, which inherits from

Starlette's `BaseHTTPMiddleware` class. The entry point for the middleware must be implemented in a function called `dispatch()`.

We use a flag to determine whether we should enable authorization. The flag is an environment variable called `AUTH_ON`, and we set it to False by default. Often when working on a new feature or when debugging an issue in our API, it's convenient to run the server locally without authorization. Using a flag allows us to switch authentication on and off according to our needs. If authorization is off, we add the default ID `test` for the request user.

Next, we check whether the user is requesting the API documentation. In that case, we don't block the request since we want to make the API documentation visible to all users, as otherwise they wouldn't know how to form their requests correctly.

> **CORS REQUESTS** CORS requests, also known as pre-flight requests, are requests sent by the web browser to understand which methods, origins, and headers are accepted by the API server. If we don't process CORS requests correctly, the web browser will abort communication with the API. Fortunately, most web frameworks contain plugins or extensions that handle CORS requests correctly. CORS requests aren't authenticated, so when we add authorization to our server, we must ensure that pre-flight requests don't require credentials.

We also check the request's method. If it's an OPTIONS request, we won't attempt to authorize the request. OPTIONS requests are preflight requests, also known as Cross-Origin Resource Sharing (CORS) requests. The purpose of a preflight request is to check which origins, methods, and request headers are accepted by the API server, and according to W3's specification, CORS requests must not require credentials (https://www.w3.org/TR/2020/SPSD-cors-20200602/). CORS requests are typically handled by the web server framework.

> **WHERE DO JSON WEB TOKENS GO?** JWTs go in the request headers, typically under the Authorization header. An Authorization header with a JWT usually has the following format: Authorization: Bearer <JWT>.

If it's not a CORS request, we attempt capture the token from the request headers. We expect the token under the Authorization header. If the Authorization header isn't found, we reject the request with a 401 (Unauthorized) status code response. The format of the Authorization header's value is `Bearer <ACCESS_TOKEN>`, so if the Authorization header is found, we capture the token by splitting header value around the space, and we attempt to validate it. If the token is invalid, PyJWT will raise an exception. In our middleware, we capture PyJWT's invalidation exceptions to make sure we can return a 401 status code response. If no exception is raised, it means the token is valid and therefore we can process the request, so we return a call to the next callback. We also store the user ID from the token payload in the request's state object so that we can access it later in the API views. Finally, to register the middleware, we use FastAPI's `add_middleware()` method.

Listing 11.4 Adding an authorization middleware to the orders API

```
from fastapi import FastAPI
from jwt import (
    ExpiredSignatureError,
    ImmatureSignatureError,
    InvalidAlgorithmError,
    InvalidAudienceError,
    InvalidKeyError,
    InvalidSignatureError,
    InvalidTokenError,
    MissingRequiredClaimError,
)
from starlette import status
from starlette.middleware.base import RequestResponseEndpoint, BaseHTTPMiddleware
from starlette.requests import Request
from starlette.responses import Response, JSONResponse

from orders.api.auth import decode_and_validate_token

app = FastAPI(debug=True)


class AuthorizeRequestMiddleware(BaseHTTPMiddleware):    #A
    async def dispatch(    #B
        self, request: Request, call_next: RequestResponseEndpoint
    ) -> Response:
        if os.getenv("AUTH_ON", "False") != "True":    #C
            request.state.user_id = "test"    #D
            return await call_next(request)    #E

        if request.url.path in ["/docs", "/openapi.json"]:    #F
            return await call_next(request)
        if request.method == "OPTIONS":
            return await call_next(request)

        bearer_token = request.headers.get("Authorization")    #G
        if not bearer_token:    #H
            return JSONResponse(
                status_code=status.HTTP_401_UNAUTHORIZED,
                content={
                    "detail": "Missing access token",
                    "body": "Missing access token",
                },
            )
        try:
            auth_token = bearer_token.split(" ")[1].strip()    #I
            token_payload = decode_and_validate_token(auth_token)    #J
        except (    #K
            ExpiredSignatureError,
            ImmatureSignatureError,
            InvalidAlgorithmError,
            InvalidAudienceError,
            InvalidKeyError,
            InvalidSignatureError,
            InvalidTokenError,
            MissingRequiredClaimError,
        ) as error:
            return JSONResponse(
```

```
                status_code=status.HTTP_401_UNAUTHORIZED,
                content={"detail": str(error), "body": str(error)},
            )
        else:
            request.state.user_id = token_payload["sub"]    #L
        return await call_next(request)


app.add_middleware(AuthorizeRequestMiddleware)    #M

from orders.api import api
```

#A We create a middleware class by inheriting from Starlette's `BaseHTTPMiddleware` base class.
#B We define the entry point for the middleware, which is an async method named `dispatch()`.
#C We authorize the request if the `AUTH_ON` environment variable is set to `True`.
#D If authorization is off, we bind a default user named `test` to the request.
#E We return by calling the next callback.
#F The documentation endpoints are publicly available, so we don't authorize them.
#G We attempt to fetch the `Authorization` header from the request object.
#H If the `Authorization` header isn't set, we return a 401 response.
#I We capture the token by splitting the contents of the `Authorizartion` header around a blank space.
#J We validate and retrieve the token's payload.
#K We capture the most common errors raised during token validation. If an error is raised, we return a 401 response.
#L We capture the user ID from the token's `sub` field.
#M We register the middle using FastAPI's `add_middleware()` method.

Our server is ready to start validating requests with JWTs! Let's run a test to see our authorization code at work. Activate the virtual environment by running `pipenv shell` and start up the server with the following command:

```
$ AUTH_ON=True uvicorn orders.web.app:app --reload
```

Now from a different terminal, make an unauthenticated request using cURL (some of the output is truncated) with the `-i` flag, which displays additional information such as the response status code:

```
$ curl -i http://localhost:8000/orders
HTTP/1.1 401 Unauthorized
[...]

{"detail":"Missing access token","body":"Missing access token"}
```

As you can see, a request with a missing token is rejected with a 401 and a message telling us that the access token is missing. Now generate a token using the jwt_generator.py script we implemented in section 11.3.3 and use the code to make a new request:

```
curl http://localhost:8000/orders -H 'Authorization: Bearer
       eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImI3NTQwM2QxLWUzZDktNDgzYy05MjZhLTM4NDRh
       M2Q4OWY1YyJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbWVzaC5pby8iLCJzdWIiOiJlYzdiYmNjZi1
       jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiLCJhdWQiOiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvb3JkZXJ
       zIiwiaWF0IjoxNjM4MTE3MjEyLjc5OTE3OSwiZXhwIjoxNjM4MjAzNjEyLjc5OTE3OSwic2NvcGUiOiJvcGV
       uaWQifQ.F1bmgYm1acfi1NMm5JGkbYQYWFNvG1-
       7BAXEnIqNdF0th_DYcnEm_p3YZ5hQ93v4QWxDx9muit6InKs-
       MHqhChP2k6DakpSocaqbgJ_IHpqNhTaEzByqZjoNfZFyQLZMo3yEaQB8S_x0LcKOOqeoPYlGSWM1eAUy7VFB
       XmvMUZrUj-yoK721U9vevgM-wdVyYFVtpTRuyjCoWMjJEVadNn-Zrxr0ghlRQnwEx-
       YdTbbEMkk_vVLWoWeEgj7mkBE167fr-
       fyGUKBqa2F71Zwh8DaDQz79Ph_STOY6BTlCnAVL8XwnlIOhJWpSHuc90Kynn_RX49_yJrQHKF-xLoflWg'
{"orders":[]}
```

If the token is valid, this time you'll get a successful response with a list of orders. Our authorization code is working! The next step is to ensure that users can only access their own resources in the server. Before we do that, though, let's add one more piece of middleware to handle CORS requests.

11.4.3 Adding CORS middleware

Since we're going to allow interactions with a frontend application, we also need to enable the CORS middleware. As we saw in section 11.4.2, CORS requests are sent by the browser to know which headers, methods, and origins are allowed by the server. FastAPI's CORS middleware takes care of populating our responses with the right information. Listing 11.5 shows how to modify the ch11/orders/app.py file to register the CORS middleware, with the newly added code in bold case and omitting some of the code in listing 11.5 with an ellipsis.

As we did previously, we use FastAPI's `add_middleware()` method to register the CORS middleware, and we pass along the necessary configuration. For testing purposes, we're using wildcards to allow all origins, methods, and headers, but in your production environment you must be more specific. In particular, you must restrict the allowed origins to your website's domain and other trusted origins.

The order in which we register our middleware matters. Middleware is executed in reverse order of registration, so the latest registered middleware is executed first. Since the CORS middleware is required for all interactions between the frontend client and the API server, we register it last, which ensures it's always executed.

Listing 11.5 Adding CORS middleware

```
import os

from fastapi import FastAPI
from jwt import (
    ExpiredSignatureError,
    ImmatureSignatureError,
    InvalidAlgorithmError,
    InvalidAudienceError,
    InvalidKeyError,
    InvalidSignatureError,
    InvalidTokenError,
    MissingRequiredClaimError,
)
from starlette import status
from starlette.middleware.base import RequestResponseEndpoint, BaseHTTPMiddleware
from starlette.middleware.cors import CORSMiddleware    #A
from starlette.requests import Request
from starlette.responses import Response, JSONResponse

from orders.api.auth import decode_and_validate_token

app = FastAPI(debug=True)


...

if os.getenv('AUTH_ON', 'False') == 'True':
    app.add_middleware(AuthorizeRequestMiddleware)


app.add_middleware(
    CORSMiddleware,    #B
    allow_origins=["*"],    #C
    allow_credentials=True,    #D
    allow_methods=["*"],    #E
    allow_headers=["*"],    #F
)


from orders.api import api
```

#A We import Starlette's `CORSMiddleware` class.
#B We register CORSMiddleware using FastAPI's `add_middleware()` method.
#C We allow all origins.
#D We support cookies for cross-origin requests.
#E We allow all HTTP methods.
#F We allow all headers.

We're almost ready! Our server can now authorize users and handle CORS requests. The next step is to ensure each user can only access their data. Move on to the next section to learn how to do that!

11.5 Authorizing resource access

We've protected our API by making sure only authenticated users can access it. Now we must ensure that the details of each order are only accessible to the user who placed it: we don't want to allow users to access each other's data. We call this type of validation authorization, and in this section, you'll learn to add it to your APIs.

11.5.1 Updating the database to link users and orders

We'll start by removing the orders currently present in the database. Those orders are not associated with a user, and therefore won't work once we enforce an association between each order and a user. `cd` into the orders directory, activate the virtual environment by running `pipenv shell`, and open a Python shell by running the `python` command. Within the Python shell, run the following code:

```
>>> from orders.repository.orders_repository import OrdersRepository
>>> from orders.repository.unit_of_work import UnitOfWork
>>> with UnitOfWork() as unit_of_work:
...     orders_repository = OrdersRepository(unit_of_work.session)
...     orders = orders_repository.list()
...     for order in orders: r.delete(order.id)
...     unit_of_work.commit()
```

Our database is now clean, so we're ready get rolling. How do we associate each order with a user? A typical strategy is to create a user table and link our orders to user records via foreign keys. But does it really make sense to create a user table for the orders service? Do we want to have a user table per service?

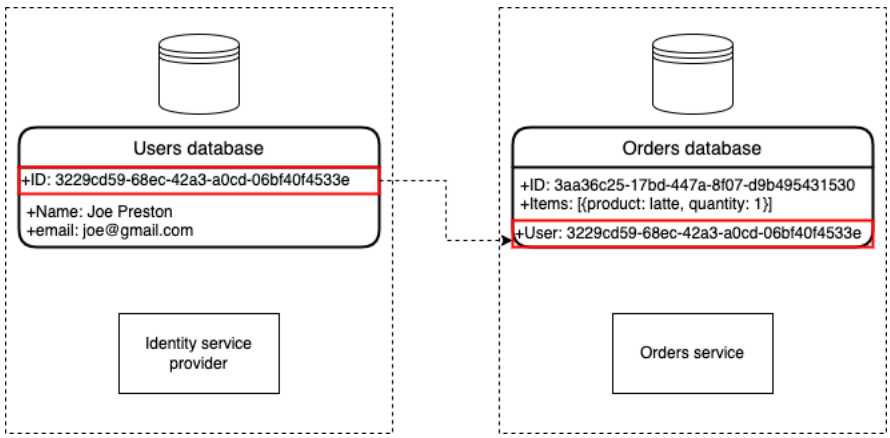


Figure 11.10 To avoid duplication, we keep only one user table under the identity service provider. And to avoid tight-coupling between services, we avoid foreign keys between the tables owned by different services.

No, we don't want to have a user table per service since it would involve lots of duplication. As you can see in figure 11.10, we want to have only one user table, and that table must be

owned by the user service. Our user service is our identity-as-a-service provider, and therefore our user table already exists. Each user has already an ID, and as we saw in section 11.3.1, the ID is present in the JWT payload under the `sub` field. So, all we need to do is add a new column to the orders table to store the ID of the user who created the order.

> **LINKING USERS TO THEIR RESOURCES** Two common anti-patterns in microservices architecture is to create one user table per service, or to have a shared user table which is directly accessed by multiple services to create foreign keys between users and other resources. Having a user table per service is unnecessary and it involves duplicates, while a shared user table across multiple services creates tight coupling between the services and risks breaking them the next time you change the user table's schema. Since JSON Web Tokens already contain opaque user IDs under the sub field, it's good practice to rely that identifier to link users to their resources.

Listing 11.6 shows how we add a `user_id` field to the `OrderModel` class. The code in listing 11.6 goes in the ch11/orders/orders/repository/models.py file, and the newly added code is highlighted with bold characters.

Listing 11.6 Adding a user ID foreign key to the order table

```
class OrderModel(Base):
    __tablename__ = 'order'

    id = Column(String, primary_key=True, default=generate_uuid)
    user_id = Column(String, nullable=False)    #A
    items = relationship('OrderItemModel', backref='order')
    status = Column(String, nullable=False, default='created')
    created = Column(DateTime, default=datetime.utcnow)
    schedule_id = Column(String)
    delivery_id = Column(String)
```

#A We add a new column called `user_id`, which is represented by a string and it's nullable.

Now that we've updated the models, we need to update the database by running a migration. As we saw in chapter 7, running a migration is the process of updating the database schema. As we did in chapter 7, we use Alembic to manage our migrations, which is Python's best database migration management library. Alembic checks the difference between the `OrderModel` model and the `order` table's current schema, and it performs the necessary updates to add the `user_id` column.

> **ALTERING TABLES IN SQLITE**. SQLite has limited support for ALTER statements. For example, SQLite doesn't support adding a new column to a table through an ALTER statement. As you can see in figure 11.11, to work around this problem in SQLite, we need to copy the table's data to a temporary table and drop the original table. Then we recreate the table with the new fields, copy back the data from the temporary table, and drop the temporary table. Alembic handles these operations with its batch operations strategy.

Before we can run the migration, we need to update the Alembic configuration. The change in listing 11.6 adds a new column to the order table, which translates into an `ALTER TABLE`

SQL statement. For local development, we're working with SQLite, which has limited support for `ALTER` statements. To ensure that Alembic generates the right migrations for SQLite, we need to update its configuration to run batch operations. **You'll only need to do this if you work with SQLite**.

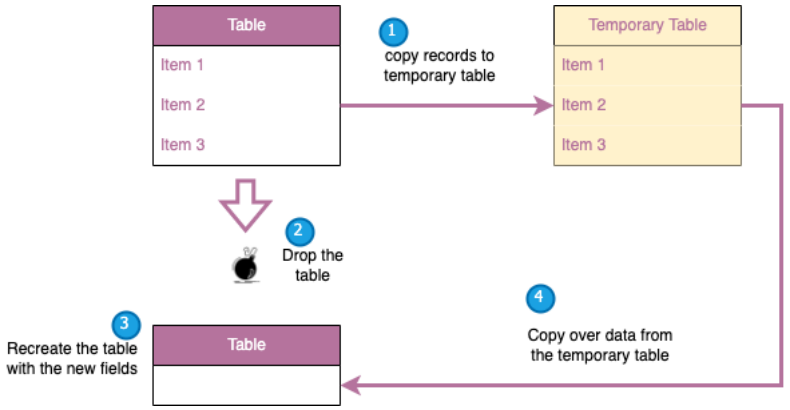


Figure 11.11 When working with SQLite, we use batch operations to make changes to our tables. In a batch operation, we copy data from the original table to a temporary table, then we drop the original table and recreate it with the new fields, and finally we copy back data from the temporary table.

To update the Alembic configuration so that we can run the migration, open the ch11/migrations/env.py file and search for a function called `run_migrations_online()`. This is the function that runs the migrations against our database. Within that function, search for the following block:

```
with connectable.connect() as connection:
    context.configure(
        connection=connection,
        target_metadata=target_metadata

    )
```

And add the following line (highlighted in bold characters) within the call to the `configure()` method:

```
with connectable.connect() as connection:
    context.configure(
        connection=connection,
        target_metadata=target_metadata,
        render_as_batch=True
    )
```

Now we can generate the Alembic migration and update the database. Run the following command to create the new migration:

```
$ PYTHONPATH=`pwd` alembic revision --autogenerate -m "Add user id to order table"
```

Next, we run the migration with the following command:

```
$ PYTHONPATH=`pwd` alembic upgrade heads
```

Our database is now ready to start linking orders and users. The next section explains how we fetch the user ID from the request object and feed it to our data repositories.

11.5.2 Restricting user access to their own resources

Now that our database is ready, we need to update our API views to capture the user ID when creating or updating an order, or when retrieving the list of orders. Since the changes that we need to make to our view functions are all quite similar, we'll illustrate how to apply the changes to some of the views. You can refer to the GitHub repository for this book for the full list of changes.

Listing 11.7 shows how to update the `create_order()` view function to capture the user ID when placing the order. The newly added code is highlighted in bold characters. As we saw in section 11.4.2, we store the user ID under the request's state property, so the first change we make is changing the signature of the `create_order()` function to include the request object. The second change is passing the user ID to the `OrderService`'s `place_order()` method.

Listing 11.7 Capturing the user ID when placing an order

```
@app.post(
    "/orders", status_code=status.HTTP_201_CREATED, response_model=GetOrderSchema
)
def create_order(request: Request, payload: CreateOrderSchema):    #A
    with UnitOfWork() as unit_of_work:
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        order = payload.dict()["order"]
        for item in order:
            item["size"] = item["size"].value
        order = orders_service.place_order(order, request.state.user_id).   #B
        unit_of_work.commit()
        return_payload = order.dict()
    return return_payload
```

#A We capture the `request` object in the `create_order()` view's signature.
#B We capture the user ID from the `request`'s `state` object.

We also need to change the `OrdersService` and the `OrdersRepository` to ensure they too capture the user ID. The below code shows how to update the `OrdersService` to capture the user ID:

```
class OrdersService:
    def __init__(self, orders_repository: OrdersRepository):
        self.orders_repository = orders_repository

    def place_order(self, items, user_id):
        return self.orders_repository.add(items, user_id)
```

And the below code shows how to update the `OrdersRepository` to capture the user ID:

```
class OrdersRepository:
    def __init__(self, session):
        self.session = session

    def add(self, items, user_id):
        record = OrderModel(
            items=[OrderItemModel(**item) for item in items], user_id=user_id
        )
        self.session.add(record)
        return Order(**record.dict(), order_=record)
```

Now that we know how to save an order with the user ID, let's see how we make sure a user only gets a list of their own orders when they call the GET /orders endpoint. Listing 11.8 shows the changes required to the `get_orders()` function, which implements the GET /orders endpoint. The newly added code is shown in bold characters. As you can see, in this case too we need to change the function's signature to capture the request object. Then we simply pass on the user ID as one of the query filters. No additional changes are required anywhere else in the code, since both the the `OrdersService` and the `OrdersRepository` are designed to accept arbitrary dictionaries of filters.

Listing 11.8 Ensuring a user only gets a list of their own orders

```
@app.get("/orders", response_model=GetOrdersSchema)
def get_orders(
    request: Request, cancelled: Optional[bool] = None, limit: Optional[int] = None
):
    with UnitOfWork() as unit_of_work:
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        results = orders_service.list_orders(
            limit=limit, cancelled=cancelled, user_id=request.state.user_id
        )
    return {"orders": [result.dict() for result in results]}
```

Let's now turn our attention to the GET /orders/{order_id} endpoint. What happens if a user tries to retrieve the details of an order that doesn't belong to them? We can respond with two strategies: we can return a 404 (Not Found) response indicating that the requested order doesn't exist, or we can respond with a 403 (Forbidden) response, indicating that the user doesn't don't have access to the requested resource.

Technically, a 403 response is more correct than a 404 when a user is trying to access a resource that doesn't belong to them. But it also exposes unnecessary information. A malicious user who's got valid credentials could leverage our 403 responses to build a map of the existing resources in the server. To avoid that problem, we'll opt for disclosing less information and return a 404 response. The user ID will become an additional filter when we attempt to retrieve an order from the database.

Listing 11.9 shows the changes required to the `get_order()` function to include the user ID in our queries, with the newly added code in bold characters. Again, we include the request object in the function signature, and we pass on the user ID to the `OrderService`'s `get_order()` method.

Listing 11.9 Filtering orders with order ID and user ID

```
@app.get("/orders/{order_id}", response_model=GetOrderSchema)
def get_order(request: Request, order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.get_order(
                order_id=order_id, user_id=request.state.user_id
            )
        return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f"Order with ID {order_id} not found"
        )
```

To be able to query orders by user ID as well, we also need to update the `OrdersService` and the `OrdersRepository` classes. We'll change their methods to accept an optional dictionary of arbitrary filters. The `OrdersService`'s `get_order()` method changes like this:

```
def get_order(self, order_id, **filters):
    order = self.orders_repository.get(order_id, **filters)
    if order is not None:
        return order
    raise OrderNotFoundError(f"Order with id {order_id} not found")
```

And the `OrdersRepository`'s `get()` and `_get()` methods require the following changes:

```
def _get(self, id_, **filters):
    return (
        self.session.query(OrderModel)
        .filter(OrderModel.id == str(id_)).filter_by(**filters)
        .first()
    )

def get(self, id_, **filters):
    order = self._get(id_, **filters)
    if order is not None:
        return Order(**order.dict())
```

The rest of the view functions in the ch11/orders/orders/web/api/api.py file require changes similar to the ones we've seen in this section, and the same goes for the remaining methods of the `OrdersService` and the `OrdersRepository` classes. As an exercise, I recommend you try to complete the changes necessary to add authorization to the remaining API endpoints. The GitHub repository for this book contains the full list of changes, so feel free to check it out for guidance and inspiration.

Our API can now authorize requests and to authorize user access to their resources. Well done on making it to this point! You're now ready to add robust authentication and authorization to our APIs! The rest of this chapter explains how to integrate with an identity as a serviced provider, and how to interact with the API with a browser application and with a microservices application.

11.6 Using an identity as a service provider

This section explains how to integrate our code with an identity as a service provider. An identity as a service provider is a service that takes care of handling user authentication and issuing access tokens for our users. Using an identity as a service provider is convenient, since it means we can focus our time and efforts on building our APIs. Good identity as a service providers are built on standards and with strong security protocols, which also reduces the security risks of our servers. In this section, you'll learn to build an integration with Auth0, which is one of the most popular identity as a service providers.

To work with Auth0, first create an account, and then create a tenant following Auth0's documentation (https://auth0.com/docs/get-started). As a first step, go to your dashboard and create an API to represent the orders API. Configure it as shown in figure 11.12, giving it http://127.0.0.1:8000/orders as the identifier's value, and selecting the RS256 signing algorithm.

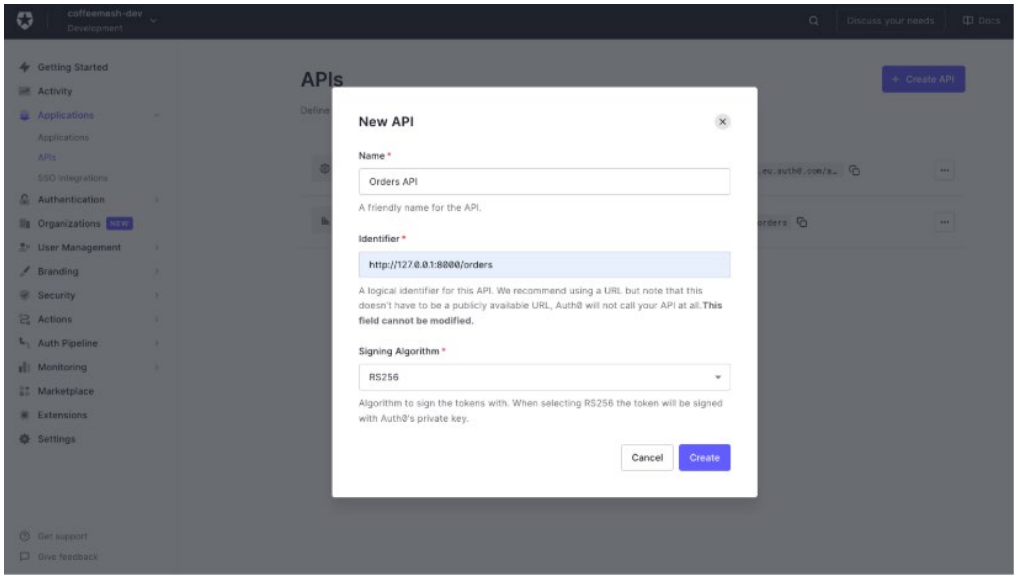


Figure 11.12 To create a new API, click the "Create API" button and fill in the form with the API's name, its URL identifier, and the signing algorithm you want to use for its access tokens.

Once you've created the API, go to "Permissions" and add a permission scope to the API, as shown in figure 11.13.

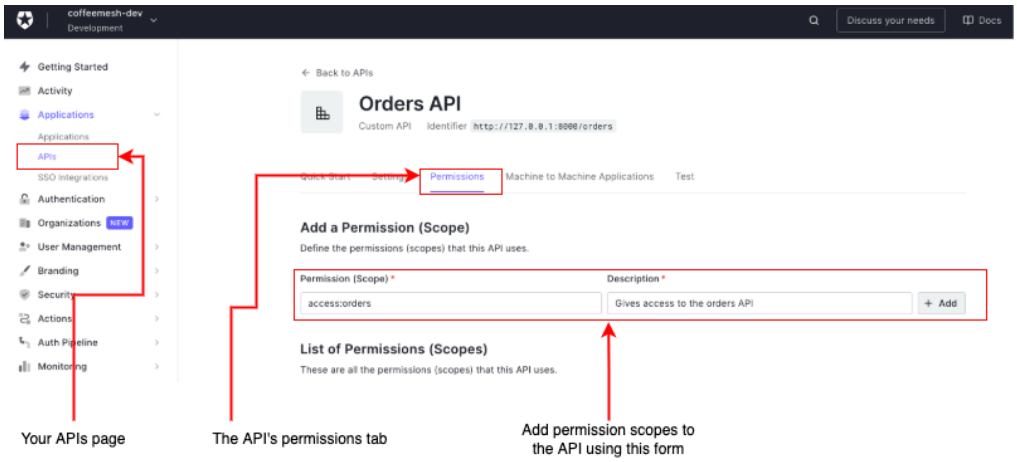


Figure 11.13 To add permission scopes to the API, click on the "Permissions" tab and fill in the "Add a Permission (Scope)" form.

Next, click on settings on the left-side bar, and then on the Custom Domains tab, as shown in figure 11.14.

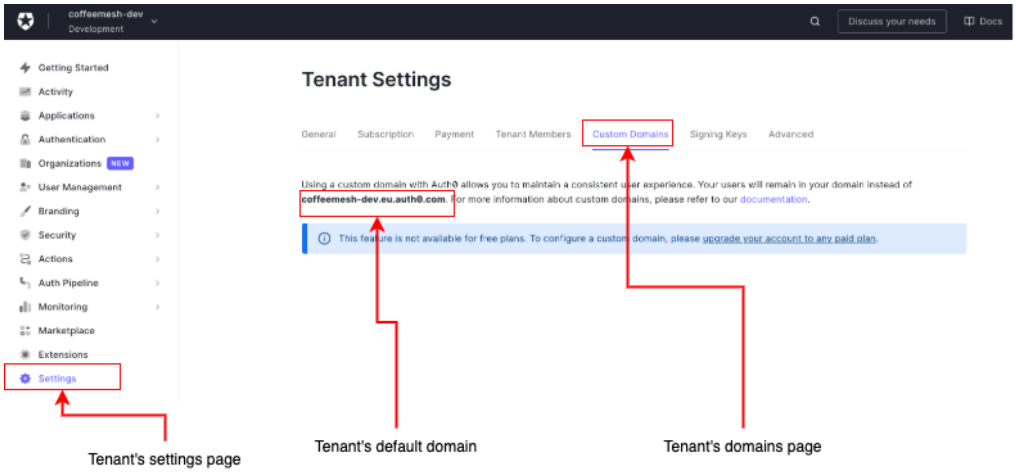


Figure 11.14 To find out your tenant's default domain, go to the tenant's settings page and click on the "Custom Domains" tab.

You can add a custom domain if you want, or you can use Auth0's default domain for your tenant. We use this domain to build the well-known URL of our authentication service:

```
htttps://<tenant>.<region>.auth0.com/.well-known/openid-configuration
```

For example, for CoffeeMesh, the tenant's domain is https://coffeemesh-dev.eu.auth0.com/.well-known/openid-configuration.

Now make a call to this URL and capture the `jwks_uri` property, which represents the URL that returns the public keys we can use to verify Auth0's tokens. For example

```
$ $ curl https://coffeemesh.eu.auth0.com/.well-known/openid-configuration | jq .jwks_uri
"https://coffeemesh-dev.eu.auth0.com/.well-known/jwks.json"
```

If you call this URL, you'll get an array of objects, each of which contains information about each of your tenant's public keys. object looks like this:

```
{
  "alg": "RS256",
  "kty": "RSA",
  "use": "sig",
  "n": "sV2z9AApyKK-
       Zo9vrzHbonNsHTgYiIOx1dHx3U102fUhPFzUcdnjb7li960iTKyTbFlMRbsN2fFZOHa5_4Q3C7UzjkVw__jK
       3AcPZ-0cCiLBS-HQzE_6ii-OPo84-
       W9Pp2ScKdAlJIqBimDtNv8vuOEMr5c5YbJz1HlppFY_hA71dgc101SHp0n9GZYqP5HV713m6smE5b7abHLqr
       USz9eVbSOrTUOcSd5_LUHvQqFb5Wt7kRalIiHnQFob-cyM1AmxDNsX1qR2cX_jqjWCRO2iK5DTG--
       ure8GQUTCMPZ0LkBKSDelTwHuEn_r4z-x30wf-2lA0yzMSlcxcJIojpQ",
  "e": "AQAB",
  "kid": "ZweIFRR4l1dJlVPHOoZqf",
  "x5t": "OJXBmAMkfObrQ9YkfUb4O20l_us",
  "x5c": [

       "MIIDETCCAfmgAwIBAgIJUbXpEMz8nlmXMA0GCSqGSIb3DQEBCwUAMCYxJDAiBgNVBAMTG2NvZmZlZW1lc2g
       tZGV2LmV1LmF1dGgwLmNvbTAeFw0yMTEwMjkyMjQ4MjBaFw0zNTA3MDgyMjQ4MjBaMCYxJDAiBgNVBAMTG2N
       vZmZlZW1lc2gtZGV2LmV1LmF1dGgwLmNvbTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALFds/Q
       AKciivmaPb68x26JzbB04GIiDsdXR8d1NdNn1ITxc1HHZ42+5YvetIkysk2xZTEW7DdnxWTh2uf+ENwu1M45
       FcP/4ytwHD2ftHAoiwUvh0MxP+oovjj6POPlvT6dknCnQJSSKgYpg7Tb/L7jhDK+XOWGyc9R5aaRWP4QO9XY
       HNdNUh6dJ/RmWKj+R1e9d5urJhOW+2mxy6q1Es/XlW0jq01DnEnefy1B70KhW+Vre5EWpSIh50BaG/nMjNQJ
       sQzbF9akdnF/46o1gkTtoiuQ0xvvrq3vBkFEwjD2dC5ASkg3pU8B7hJ/6+M/sd9MH/tpQNMszEpXMXCSKI6U
       CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zAdBgNVHQ4EFgQUWrl+q/l4wp/MWDdYrhjxns0iP2wwDgYDVR0
       PAQH/BAQDAgKEMA0GCSqGSIb3DQEBCwUAA4IBAQA+YH+sxcMlBzEOJ5hJgZw1upRroCgmeQzEh+Cx73sTKw+
       vi8u70bdkDt9sBLKlGK9xbPJt3+QWZDJF9rwx4vXbfFvxZD+dthIvn4NH4/sLQXG20JN/b6GtHdVllbJIGUe
       Wb8DBsx94wXYMwag0gXUk5spgaGGdoc16uSrrbxt/rmzFk3VMQ8qG5i8E33N/DZb88P4u3WJMNMsmujw9Q8m
       eg4ygEFadXBcfJPHuiriLWi0j1Gm+m6DZQM51OtpQ/cvcZXRNPogqj7wsZXH4za9DJjnQf8ZOKQ86WKl/9CE
       5AvHBTTTr810DviJIqv8sqC866+2t2euxcfOYMIw5E42o"
  ]
}
```

The two most important fields in this payload are `kid` and `x5c`. `kid` is the ID of the key, and we use it to match the `kid` field of the JWT's header section. It tells us which key we need to use to verify the token's signature. The `x5c` field contains an array of public keys in the form of X.509 certificates, the first of which we use to verify the JWT's signature.

This is all the information we need to integrate our code with Auth0. We'll implement our Auth0 integration in the ch11/orders/web/api/auth.py module, which we created earlier in section 11.4.1 to encapsulate our authorization code. Go ahead and delete the contents of ch11/orders/web/api/auth.py, and replace them with the contents of listing 11.10. We first import the necessary dependencies, create a template for the X.509 certificate, and load the public keys from the well-known endpoint. X.509 certificates are wrapped between a `-----BEGIN CERTIFICATE-----` and a `-----END CERTIFICATE-----` statements, so our template

includes both statements together with a template variable named key, which we'll use to substitute the actual key.

Since Auth0 uses several keys to sign the tokens, we load the public keys by calling the JWKS endpoint and we dynamically load the right key for the given token. As you can see in figure 11.15, the `kid` property in the token's headers tells us which key we need to use, and our custom function `_get_certificate_for_kid()` finds the X.509 certificate for the token's `kid`. To load the key, we use cryptography's `load_pem_x509_certificate()` function, passing in the public key formatted into our X.509 certificate byte-encoded.

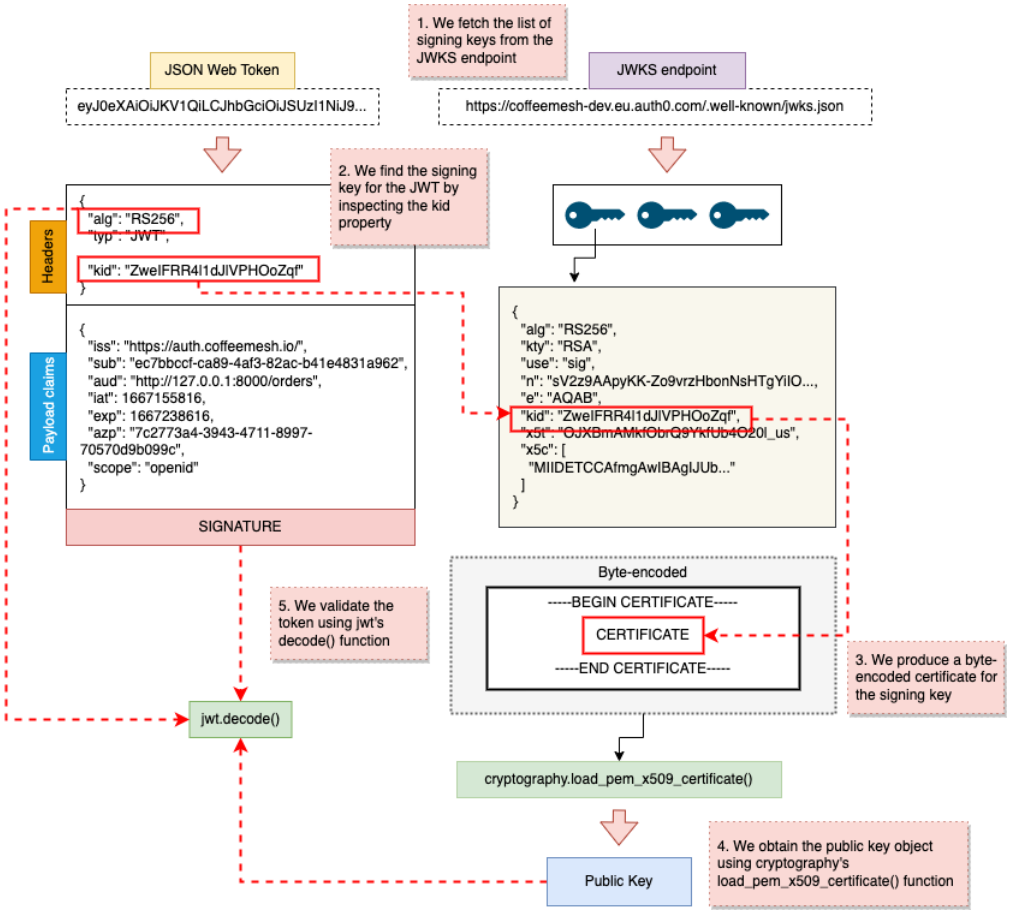


Figure 11.15 To validate a JWT, we verify its signature using its corresponding signing key. The signing key is available in the JWKS endpoint.

Since tokens can be signed with different algorithms, we fetch the algorithms directly form the token's headers. Auth0 issues tokens that can access both our API and the user information API, so we include both services in the audience.

Listing 11.10 Adding an authorization module to the API

```
import jwt
import requests
from cryptography.x509 import load_pem_x509_certificate


X509_CERT_TEMPLATE = "-----BEGIN CERTIFICATE-----\n{key}\n-----END CERTIFICATE-----"    #A


public_keys = requests.get(
    "https://coffeemesh-dev.eu.auth0.com/.well-known/jwks.json"
).json()["keys"]    #B



def _get_certificate_for_kid(kid):    #C
    """
    Return the public key whose ID matches the provided kid.
    If no match is found, an exception is raised.
    """
    for key in public_keys:
        if key["kid"] == kid:    #D
            return key["x5c"][0]
    raise Exception(f"Not matching key found for kid {kid}")    #E


def load_public_key_from_x509_cert(certificate):    #F
    """
    Loads the public signing key into a RSAPublicKey object. To do that,
    we first need to format the key into a PEM certificate and make sure
    it's utf-8 encoded. We can then load the key using cryptography's
    convenient `load_pem_x509_certificate` function.
    """
    return load_pem_x509_certificate(certificate).public_key()    #G

def decode_and_validate_token(access_token):    #H
    """
    Validates an access token. If the token is valid, it returns the token payload.
    """
    unverified_headers = jwt.get_unverified_header(access_token)    #I
    x509_certificate = _get_certificate_for_kid(unverified_headers["kid"])    #J
    public_key = load_public_key_from_x509_cert(
        X509_CERT_TEMPLATE.format(key=x509_certificate).encode("utf-8")    #K
    )
    return jwt.decode(    #L
        access_token,
        key=public_key,
        algorithms=unverified_headers["alg"],    #M
        audience=[    #N
            "http://127.0.0.1:8000/orders",
            "https://coffeemesh-dev.eu.auth0.com/userinfo",
        ],
    )
```

#A We define a string template for a X509 certificate where we can substitute the contents of the certificate's key.
#B We pull the list of signing keys from the tenant's well-known endpoint.
#C We define a function that returns the certificate for a given key ID.
#D We iterate our tenant's array of keys, and if one of them matches the key ID supplied, we return the certificate.
#E If a match isn't found, we raise an exception.
#F We define a function that loads the public key object for a given certificate.
#G We load the public key using cryptography's `load_pem_x509_certificate()` function.
#H We define a function that decodes and validates a JSON Web Token.
#I We fetch token's headers without verification.
#J We fetch the certificate corresponding to the token's key ID.
#K We load the certificate's public key object.
#L We validate and decode the token using jwt's `decode()` function.
#M We verify the token's signature using the algorithm indicated in the token's header.
#N We pass the list of expected audiences for the token.

Ready to go! The orders service is now able to validate tokens issued by Auth0. The following sections illustrate how the leverage this integration to make our API server accessible to a Single Page Application (SPA) and to another microservice.

11.7 Using the implicit authorization flow

In the implicit flow the API client requests an ID token and an access token directly from the authorization server. As we explained in section 11.2.2, we must use the access token to interact with the API server. The ID token can be used in the UI to show the details of the user, but it must never be sent to our API server.

To illustrate how this flow works, we need to use an SPA, so the directory for this chapter in the GitHub repository for this book includes a simple UI built with Vue.js. The UI is located under the ch11/ui folder.

We'll first configure the application. Go to your Auth0 account and create a new application. Select Single Page Web Applications and give it a name, then click "Create". In the applicatin's settings page, under the Application URIs section, give the value of http://localhost:8000 to the "Allowed Callback URLs", the "Allowed Logout URLs", the "Allowed Web Origins", and the "Allowed Origins (CORS)" fields. From the application's settings, we need two values to configure our application: the Domain and the Client ID. Open the ch11/ui/.env.local file and replace the value for `VUE_APP_AUTH_CLIENT_ID` with the client ID, and `VUE_APP_AUTH_DOMAIN` with the Domain from your Auht0 application's settings page.

To run the UI, you'll need an up-to-date version of node.js and npm, which you can download from the node.js website (https://nodejs.org/en/). Once you've installed node.js and npm, you'll need to install yarn with the following command:

```
$ npm install -g yarn
```

Next, cd into the ch11/ui folder and install the dependencies by running the following command:

```
$ yarn
```

Once the application is configured, you can run it by executing the following command:

```
$ yarn serve --mode local
```

The application will become available under the http://localhost:8080 address. Make sure the orders API is also running, since the Vue.js application talks to it. To run the orders API, run the following command from the ch11/orders folder:

```
$ AUTH_ON=True uvicorn orders.web.app:app --reload
```

Once you register a user through the UI, you'll be able to see your authorization token in the UI. You can use this token to call the API directly from the terminal. For example, you can get a list of orders for your user by calling the API with the following command:

```
$ curl http://localhost:8000/orders -H 'Authorization: Bearer <ACCESS_TOKEN>'
```

Through the Vue.js application, you can create new orders and display the orders placed by the user by clicking the "Show my orders" button.

The implicit authorization flow works for users accessing your APIs through the browser. However, this flow isn't convenient for machine-to-machine communication. To allow more programmatic access to your APIs, you need to support the client credentials flow. In the next section, we explain how to enable that flow!

11.8 Using the client credentials flow

This section explains how to implement the client credentials flow for server-to-server communication. We use the server-to-server flow when we must authenticate our own services to access other APIs, or when we want to allow programmatic access to our APIs. In the client credentials flow our services request an access token from the authentication service by providing a shared secret together with the client ID and the desired audience. We can then use this access token to access the API of the target audience.

To use this authorization flow, you need to register a server-to-server client with your identity-as-a-service provider. In your Auth0 dashboard's applications page, click "Create Application", and select "Machine to Machine Applications". Give it a name and click "Create". In the next screen, where you're asked to select the API you want to authorize this client for, select the orders API, and then select the permission we created in section 11.6. Once you've registered the client, you'll get a client ID and a client secret, which you can use to obtain access tokens.

Listing 11.11 shows how use implement server-to-server authorization to obtain an access token and make a call to the orders API. The code in listing 11.11 is available in the book's GitHub repository under the ch11/machine_to_machine_test.py file. We create a function to obtain the access token from the authorization server by calling the POST https://coffeemesh-dev.eu.auth0.com/oauth/token endpoint. In the payload, we provide the client ID and the client secret, and we specify the audience for which we want to generate the access token. We also declare that we want to use the client credentials flow under the `grant_type` property. If the client is correctly authenticated, we get back an access token which we then use to call the orders API.

Listing 11.11 Authorizing a client for machine-to-machine access to the orders API

```
import requests


def get_access_token():
    payload = {
        "client_id": "MQjOO2o3rdM3XhoiTB5cr497Irr63g8n",
        "client_secret": "gU5CsCxcFQT3RnJXupaMPsg6-
       PwzPiwo1Q0K0Pg16fLkWG1AdgXjGasZoYNkP2_j",
        "audience": "http://127.0.0.1:8000/orders",
        "grant_type": "client_credentials"
    }

    response = requests.post(
        "https://coffeemesh-dev.eu.auth0.com/oauth/token",
        json=payload,
        headers={'content-type': "application/json"}
    )

    return response.json()['access_token']


def create_order(token):
    order_payload = {
        'order': [{
            'product': 'capuccino',
            'size': 'small',
            'quantity': 1
        }]
    }

    order = requests.post(
        'http://127.0.0.1:8000/orders',
        json=order_payload,
        headers={'content-type': "application/json", "authorization": f"Bearer {token}"}
    )

    return order.json()


access_token = get_access_token()
order = create_order(access_token)
print(order)
```

This concludes our journey through API authentication and authorization, and what a journey! You've learned about Open Authorization, OpenID Connect and the available authorization flows. You've learned what JSON Web Tokens are, how to inspect their payloads, and how to produce and validate them. You've learned how to add authentication and authorization to your APIs, and how to interact with the API using the implicit and the client credentials flows. You've got all you need to start adding robust authentication and authorization to your own APIs!

11.9 Summary

- The recommended way to add authentication and authorization to our APIs is using the standard protocols Open Authorization and OpenID Connect.
- **Open Authorization** is an access delegation protocol that allows a user to grant an application access to resources they own in a different website.
- Open Authorization distinguishes four **authorization flows**:
    - **Authorization code**: in this flow, the API server exchanges a code with the authorization server to request the user's access token.
    - **Implicit**: in this flow, the client application, typically an SPA, requests the user access token directly from the authorization server.
    - **Resource owner password**: in this flow, the user sends their credentials to the API server to grant access to their account in a different website.
    - **Client credentials**: in this flow, the client, typically another microservice, exchanges a private secret in return for an access token.
- **OpenID Connect** is an identity verification protocol that builds on top of Open Authorization. It helps users to easily authenticate to new websites by bringing their identity from other websites, such as Google or Facebook.
- **JSON Web Tokens (JWT)** are JSON documents which contain claims about the user's access permissions. JWTs are encoded using base64url encoding, and are typically signed using a private/public key.
- To authenticate a request, users send their access tokens in the request's Authorization header. The expected format of this header is: `Authorization: Bearer <ACCESS_TOKEN>`.
- We use PyJWT to validate access tokens. PyJWT checks that the token isn't expired, that the audience is correct, and that the signature can be verified with one of the available public keys. If the token is invalid, we reject the request with a 401 (Unauthorized) response.
- To link users to their resources, we use the user ID as represented in the `sub` claim of the JWT.
- If a user tries to access a resource that doesn't belong to them, we respond with a 403 (Forbidden) response.
- OPTIONS requests are known as **CORS** requests or **pre-flight** requests. CORS requests must not be protected by credentials.