**Part 1. Introducing Microservice APIs**

Microservices are an architectural style in which components of a system are designed as standalone and independently deployable applications. The concept of microservices has been around since the early 2000s, and since the 2010s it has gained in popularity. Nowadays, microservices are a popular choice for building modern websites. As you’ll learn in chapter 1, microservices allow you to leverage the power of distributed applications, scale components more easily, and release faster.

However, for all their benefits, microservices also come with challenges of their own. They bring a substantial infrastructure overhead, and they’re more difficult to monitor, operate, and trace. When working with microservices, the first challenge is to get their design right, and in chapter 3 you’ll learn several principles and strategies that will help you build robust microservices.

Microservices collaborate through APIs, and in this book, you’ll learn to design and build REST and GraphQL APIs for your microservices. Chapter 2 gives you a taste of building a REST API, and in the second part of this book, you’ll learn additional patterns and principles to build robust REST APIs. The most challenging aspect of working with APIs is ensuring that both the API client and the API server follow the API specification, and in chapter 1 you’ll learn about documentation-driven development and the importance of starting the API journey with a good and well-documented design.

# 1 What are microservice APIs?

This chapter covers

* What microservices are and how they compare with monolithic applications
* What web APIs are and how they help us drive integrations between microservices
* The most important challenges of developing and operating microservices

This chapter defines the most important concepts in this book: microservices and APIs. Microservices are an architectural style in which components of a system are designed as independently deployable services, and APIs are the interfaces that allow us to interact with those services. We will see the defining features of microservices architecture and how they compare with monolithic applications. Monolithic applications are structured around a single code base and deployed in a single build.

We’ll discuss the benefits and the disadvantages of microservices architecture. The last part of this chapter talks about the most important challenges that we face when designing, implementing, and operating microservices. This discussion is not to deter you from embracing microservices, but so that you can make an informed decision about whether microservices are the right choice of architecture for you.

## 1.1 What are microservices?

In this section, we define what microservices architecture is, and we analyze how microservices compare with monolithic applications. We’ll look into the benefits and challenges of each architectural pattern. Finally, we’ll also take a brief look at the historical developments that led to the emergence of modern microservices architecture.

### 1.1.1 Defining microservices

So, what are microservices? Microservices can be defined in different ways, and, depending on which aspect of microservices architecture we want to emphasize, authors provide slightly different yet related definitions of the term. 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**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1070924)

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 that microservices are “small.” In this context, “small” doesn’t refer to the size of the microservices’ code base, but to the idea 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. They define microservices as an 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” (<https://martinfowler.com/articles/microservices.html>). This definition emphasizes the autonomy of the services by stating that they run in 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 communicate through lightweight protocols, such as HTTP.

**DEFINITION** A *microservice* is an architectural style in which components of a system are designed as independently deployable services. Microservices are designed around well-defined business subdomains, and they talk to each other using lightweight protocols, such as HTTP.

From the previous definitions, we can see that microservices can be defined as an architectural style in which services are components that perform a small and clearly defined set of related functions. As you can see in figure 1.1, this definition means that a microservice is designed and built around a specific business subdomain, for example, processing payments, sending emails, or handling orders from a customer.

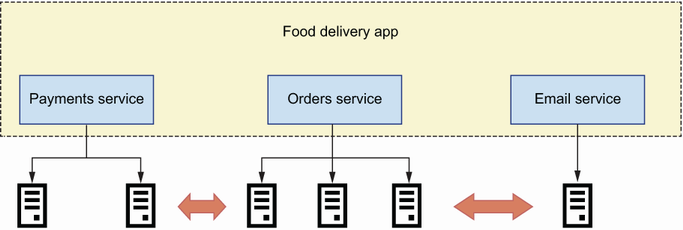


Figure 1.1 In microservices architecture, every service implements a specific business subdomain and is deployed as an independent component that runs in its own process.

Microservices are deployed as independent processes, typically running in independent environments, and expose their capabilities through well-defined interfaces. In this book, you will learn to design and build microservices that expose their capabilities through web APIs, though other types of interfaces are also possible, such as messaging queues.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1070939)

### 1.1.2 Microservices vs. monoliths

Now that we know what microservices are, let’s see how they compare with the monolithic application pattern. In contrast with microservices, a monolith is a system where all functionality is deployed together as a single build and runs in the same process. For example, figure 1.2 shows a food delivery application with four services: a payments service, an orders service, a delivery service, and a customer support service. Since the application is implemented as a monolith, all functionality is deployed together. We can run multiple instances of a monolithic application and have them run in parallel for redundancy and scalability purposes, but it’s still the whole application running in each process.

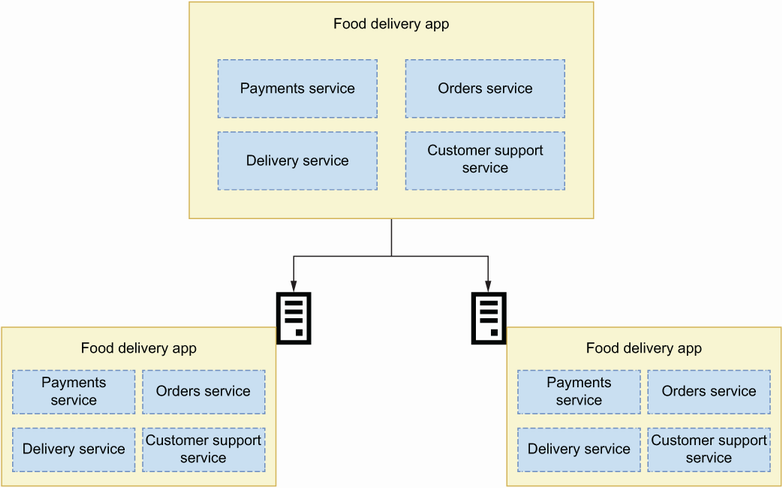


Figure 1.2 In a monolithic application, all functionality is deployed together as a single build to each server.

**DEFINITION** A *monolith* is an architectural pattern in which the whole application is deployed as a single build.

In some situations, the monolith is the right choice of architecture. For example, we’d use a monolith when our code base is small and it isn’t expected to grow very large.[**3**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1070953) Monoliths also come with advantages. First, having the whole implementation in the same code base makes it easier to access data and logic from different subdomains. And because everything runs within the same process, it is easy to trace errors through the application: you only need to place a few breakpoints in different parts of your code, and you will get a detailed picture of what happens when something goes wrong. Besides, because all the code falls within the scope of the same project, you can leverage the productivity features of your favorite development editor when consuming functionality from a different subdomain.

However, as the application grows and becomes more complex, this type of architecture shows limitations. This happens when the code base grows to a point where it becomes difficult to manage, and when finding your way through the code becomes arduous. Additionally, being able to reuse code from other subdomains within the same project often leads to tight coupling among components. Tight coupling happens when a component depends on the implementation details of another piece of code.

The bigger the monolith, the longer it takes to test it. Every part of the monolith must be tested, and as we add new features to it, the test suite grows larger. Consequently, deployments become slower and encourage developers to pile up changes within the same release, which makes releases more challenging. Because many changes are released together, if a new bug is introduced in the release, it is often difficult to spot the specific change that caused the bug and roll it back. And because the whole application runs within the same process, when you scale the resources for one component, you are scaling for the whole application. Long story short, code changes become increasingly risky and deployments become more difficult to manage. How can microservices help us address these issues?

Microservices address some of the issues associated with monolithic applications by enforcing strict boundaries separating components. When you implement an application using microservices, each microservice runs in a different process, often in different servers or virtual machines, and can have a completely different deployment model. As a matter of fact, they can be written in completely different programming languages (that does not mean they should!).

Because microservices contain smaller code bases than a monolith, and because their logic is self-contained and defined within the scope of a specific business subdomain, 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 code is clearer, and it is easier to refactor them. This means the code can get better over time and become more maintainable. Consequently, we can make small changes to the code and release more often. Smaller releases are more controllable, and if we spot a bug, the releases are easier to roll back. I’d like to emphasize that microservices are not a panacea. As we will see in section 1.3, microservices also have limitations and bring challenges of their own.

Now that we know what microservices are and how they compare with monolithic applications, let’s take a step back and see what developments led to the emergence of this type of architecture.

### 1.1.3 Microservices today and how we got here

In many ways, microservices are not new.[**4**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1070965) Companies were 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 code base 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. To combat these issues, they decided to look for independent pieces of logic within the code and separate 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 the data except through an API. They called this new type of architecture *service-oriented architecture* (<https://vimeo.com/29719577>). Netflix also pioneered this type of architectural style at scale, and they referred to it as “fine-grained Service Oriented Architecture.”[**5**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1070970)

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” (<https://vimeo.com/74452550>). In 2014 the concept was consolidated with a paper written by Martin Fowler and James Lewis about the architectural features of microservices (<https://martinfowler.com/articles/microservices.html>), as well as the publication of Newman’s influential book *Building Microservices*.

Today, microservices are a widely used architectural style. Most companies in which technology plays an important role are already using microservices or moving toward its adoption. It is also common for startups to begin implementing their platform using a microservices approach. However, microservices are not for everyone, and although they bring substantial benefits, as we have shown, they also carry considerable challenges, as we will see in section 1.3.

## 1.2 What are web APIs?

In this section, we will explain web APIs. You will learn that a web API is a specific instance of the more general concept of an 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 different types of interfaces. For this reason, we will begin this section by defining what an API is, and then we will move on to explaining how APIs help us drive integrations between microservices.

### 1.2.1 What is an API?

An API is an interface that allows us to programmatically interact with an application. Programmatic interfaces are those we can use from our code or from the terminal, as opposed to graphic interfaces, in which we use a user interface to interact with the application. There are multiple types of application interfaces, such as command-line interfaces (CLIs; interfaces that allow you to use an application from a terminal), desktop UI interfaces, web UI interfaces, or web API interfaces. As you can see in figure 1.3, an application can have one or more of these interfaces.

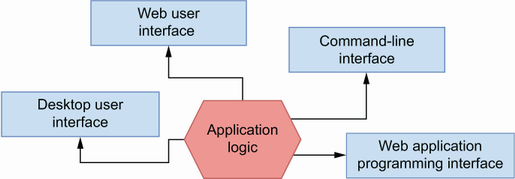


Figure 1.3 An application can have multiple interfaces, such as a web API, a CLI, a web UI, and a desktop UI.

To illustrate this idea, think of the popular client URL (cURL). cURL is a CLI to the libcurl library. libcurl implements functionality that allows us to interact with URLs, while cURL exposes those capabilities through a CLI. For example, we can use cURL to send a GET request to a URL:

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

We can also use cURL with the -O option in order to download the contents of a URL to a file:

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

The libcurl library sits behind the cURL CLI, and nothing prevents us from accessing it directly through the source code (if you are curious, you can pull it from Github: <https://github.com/curl/curl>) and 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. A web API is an API that uses the Hypertext Transfer Protocol (HTTP) protocol to transport data. HTTP is the communication protocol that underpins the internet, and it allows us to exchange different kinds of media types, such as text, images, video, and JSON, over a network. HTTP uses the concept of a Uniform Resource Locator (i.e., URL) to locate resources on the internet, and it has features that can be leveraged by API technologies to enhance the interaction with the server, such as request methods (e.g., GET, POST, PUT) and HTTP headers. Web APIs are implemented using technologies such as SOAP, REST, GraphQL, gRPC, and others that are discussed in more detail in appendix A.

### 1.2.3 How do APIs help us drive microservices integrations?

Microservices communicate with each other using APIs, and therefore APIs represent the interfaces to our microservices. APIs are documented using standard protocols. The API documentation tells us exactly what we need to do to interact with the microservice and what kind of responses we can expect from it. The better the API documentation, the clearer it is for the API consumer how the API works. In that sense, as you can see in figure 1.4, API documentation represents a contract between services: as long as both the client and the server follow the API documentation, communication will work as expected.

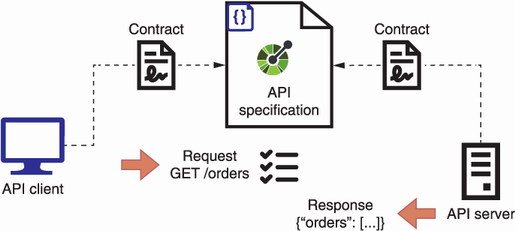


Figure 1.4 API specifications represent a contract between the API server and the API client. As long as both the client and the server follow the specification, the API integration will work.

Fowler and Lewis popularized the idea that the best strategy for integrating microservices is by exposing *smart endpoints* and communicating through *dumb pipes* (<https://martinfowler.com/articles/microservices.html>). This idea is inspired by the design principles of Unix systems, which establish that

* A system should be made up of small, independent components that 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 are simple mechanisms for passing messages from one application to another. To illustrate this process, think of the following chain of commands, which you can run from the terminal of a Unix-based machine (e.g., a Mac or Linux computer):

$ history | less

The history command shows you the list of all commands you have run using your Bash profile. The list of commands can be long, so you may want to paginate history’s output using the less command. To pass data from one command to the another, use the pipe character (|), which instructs the shell to capture the output from the history command and pipe it as 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, web APIs exchange data through HTTP. The data transport layer knows nothing about the specific API protocol we are using, and therefore it represents our “dumb pipe,” while the API itself contains all the necessary logic to process the data.

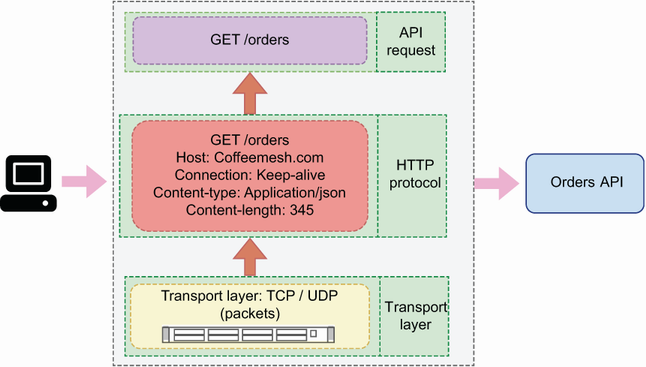


Figure 1.5 Microservices communicate over APIs using a data transport layer, such as HTTP over TCP.

APIs must be stable, and behind them you can change the internal implementations of any service provided they comply with the API documentation. This means that the consumer of an API must be able to continue calling the API in the exact way as before, and it must get the same responses. This leads to another important concept in microservices architecture: *replaceability*.[**6**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1071051) The idea is that you should be able to completely replace the code base that lies behind an endpoint, yet the endpoint, and therefore communication across services, will still work. Now that we understand what APIs are and how they help us drive integrations between services, let’s look at the most important challenges posed by microservices.

## 1.3 Challenges of microservices architecture

As we saw in section 1.1.2, microservices bring substantial benefits. However, they also come with significant challenges. In this section, we discuss the most important challenges that microservices pose, which we classify into five main categories:

* Effective service decomposition
* Microservices integration tests
* Handling service unavailability
* Tracing distributed transactions
* Increased operational complexity and infrastructure overhead

All the problems and difficulties that we discuss in this section can be addressed with specific patterns and strategies, some of which we detail over the course of this book. You’ll also find references to other resources that deal with these issues in depth. The idea here is to make you aware that microservices are not a magical cure for all the problems that monolithic applications present.

### 1.3.1 Effective service decomposition

One of the most important challenges when designing microservices is service decomposition. We must break down a 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 between services is not resilient, or there are enough dependencies between both components to justify merging them. Failing to break down a system into independent microservices can result in what Chris Richardson, author of *Microservices Patterns*, 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. In chapter 3, you’ll learn useful design patterns and service decomposition strategies that will help you break down a system into microservices.

### 1.3.2 Microservices integration tests

In section 1.1.2, we said that microservices are usually easier to test, and that their test suites generally run faster. Microservices integration tests, however, can be significantly more difficult to run, especially in cases where a single transaction 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, to test the integration among multiple services you need to be able to run all of them with a setup similar to your 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 test the API implementation against the API specification using tools like Dredd and Schemathesis, as you’ll learn in chapter 12. 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.[**7**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1071081) Finally, none of these tests will be sufficient without a full-blown end-to-end test that runs the actual microservices making calls to each other.

### 1.3.3 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 we place an order through a food delivery application such as Uber Eats, Delivery Hero, or Deliveroo, 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 take a high-level view of the process that takes place when a user places an order (see figure 1.6 for an illustration of the chain of requests):

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

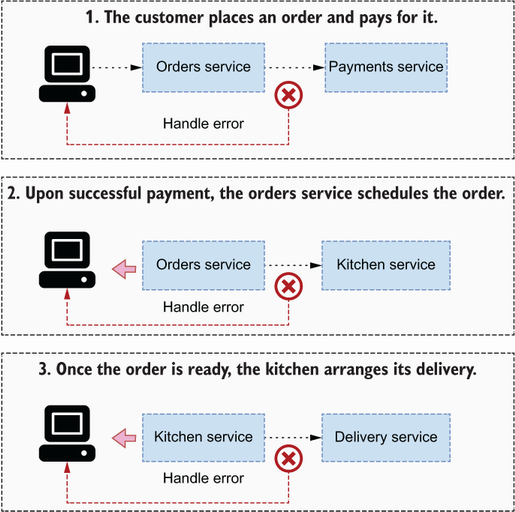


Figure 1.6 Microservices must be resilient to events such as service unavailability, request timeouts, and processing errors from other services and either retry the requests or come back to the user with a meaningful response.

In this complex chain of requests, if one of the services involved fails to respond as expected, it can trigger a cascading error through the platform that leaves the order unprocessed or in an inconsistent state. For this reason, it is important to design microservices so that they can deal reliably with failing endpoints. Our end-to-end tests should consider these scenarios and test the behavior of our services in those situations.

### 1.3.4 Tracing distributed transactions

Collaborating services must sometimes handle distributed transactions. Distributed transactions are those that require the collaboration of two or more services. For example, in a food delivery application, we want to keep track of the existing stock of ingredients so that our catalogue can accurately reflect product availability. When a user places an order, we want to update the stock of ingredients to reflect the new availability. Specifically, we want to update the stock of ingredients once the payment has been successfully processed. As you can see in figure 1.7, the successful processing of an order involves the following actions:

1. Process the payment.
2. If payment is successful, update the order’s status to indicate that it’s in progress.
3. Interface with the kitchen service to schedule the order for production.
4. Update the stock of ingredients to reflect their current availability.

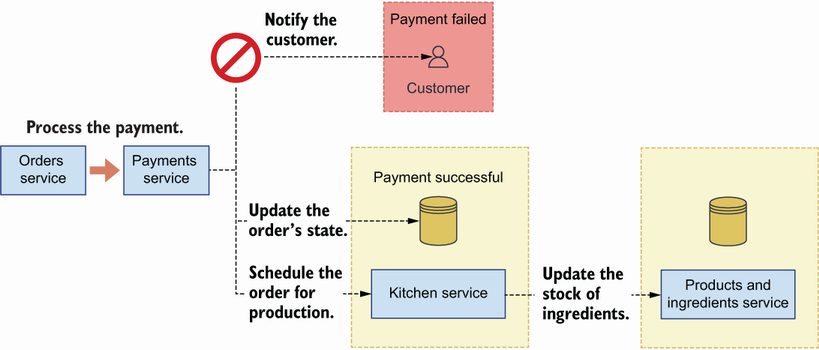


Figure 1.7 A distributed transaction involves collaboration among multiple services. If any of these services fails, we must be able to handle the failure and provide a meaningful response to the user.

All of these operations are related, and they must be orchestrated so that they either all succeed or fail 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 happen within the same process, managing the flow is straightforward, but with microservices we must manage the outcomes of various processes. When using microservices, the challenge is ensuring that we have a robust communication process among services so that we know exactly what kind of error happens when it does, and we take appropriate measures in response to it.

In the case of services that work collaboratively to serve certain requests, you also must be able to trace the cycle of the request as it goes across the different services to be able to spot errors during the transaction. To gain visibility of distributed transactions, you’ll need to set up distributed logging and tracing for your microservices. You can learn more about this topic from Jamie Riedesel’s *Software Telemetry* (Manning, 2021).

### 1.3.5 Increased operational complexity and infrastructure overhead

Another important challenge that comes with microservices is the increased operational complexity and operational overhead they add to your platform. When the whole backend of your website runs within a single application build, you only need to deploy and monitor one process. When you have a dozen microservices, every service must be configured, deployed, and managed. And this includes not only the provisioning of servers to deploy the services, but also log aggregation streams, monitoring systems, alerts, self-recovery mechanisms, and so on. As you’ll learn in chapter 3, every service owns its own database, which means they also require multiple database setups with all the features needed to operate at scale. And it is not unusual that a new deployment changes the endpoint for a microservice, whether it’s the IP, the base URL, or a specific path within a generic URL, which means its consumers must be notified of the changes.

When Amazon first started their journey toward a microservices architecture, they discovered that development teams would spend about 70% of their time managing infrastructure (<https://vimeo.com/29719577> at 07:53). 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.

## 1.4 Introducing documentation-driven development

As we explained in section 1.2.3, the success of an API integration depends on good API documentation, and in this section, we introduce an API development workflow that puts documentation at the forefront of API development. As you can see in figure 1.8, documentation-driven development is an approach to building APIs that works in three stages:

1. You design and document the API.
2. You build the API client and the API server following the documentation.
3. You test both the API client and the API server against the documentation.

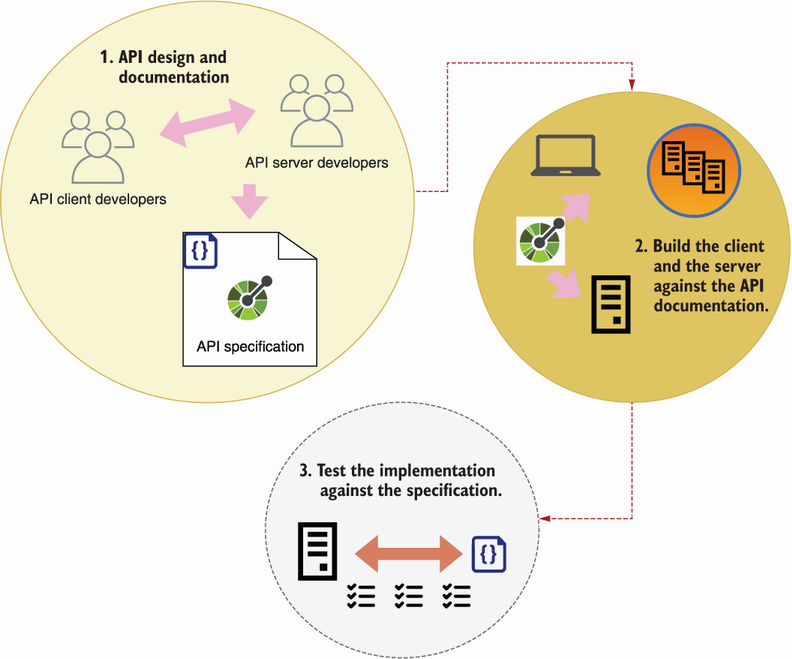


Figure 1.8 Documentation-driven development works in three stages: design and document, implement, and validate.

Let’s dive into each of these points. The first step involves designing and documenting the specification. We build APIs for others to consume, so before we build the API, we must produce an API design that meets the needs of our API clients. Just as we involve users when we design an application’s user interface (UI), we must also engage with our API consumers when we design the API.

Good API design delivers good developer experience, while good API documentation helps to deliver successful API integrations. What is API documentation? API documentation is a description of the API following a standard interface description language (IDL), such as OpenAPI for REST APIs and the Schema Definition Language (SDL) for GraphQL APIs. Standard IDLs have ecosystems of tools and frameworks that make it easier to build, test, and visualize our APIs, and therefore it’s worth investing time in studying them. In this book, you’ll learn to document your APIs with OpenAPI (chapter 5) and the SDL (chapter 8).

Once we have produced a documented API design, we move on to the second stage, which consists of building the API server and the API client against the API documentation. In chapters 2 and 6, you’ll learn to analyze the requirements of an OpenAPI specification and to build an API application against them, and in chapter 10, we’ll apply the same approach to GraphQL APIs. API client developers can also leverage the API documentation to run API mock servers and test their code against them.[**8**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/01.htm#pgfId-1071148)

The final stage involves testing our implementation against the API documentation. In chapter 12, you’ll learn to use automated API testing tools such as Dredd and Schemathesis, which can generate a solid battery of tests for your API. Running Dredd and Schemathesis in combination with your application unit test suite will give you confidence that your API implementation works as it should. You should run these tests in your continuous integration server to make sure you don’t release any code that breaks the contract with the API documentation.

By putting API documentation at the forefront of the development process, documentation-driven development helps you avoid one of the most common problems API developers face: disagreements between the client and the server development teams about how the API should work. In the absence of robust API documentation, developers often need to guess on implementation details of the API. In such cases, the API rarely succeeds its first integration test. Although documentation-driven development won’t give a 100% guarantee that your API integrations will work, it will significantly reduce the risk of API integration failure.

## 1.5 Introducing the CoffeeMesh application

To illustrate the concepts and ideas that we explain throughout this book, we’ll build components of an application called CoffeeMesh. CoffeeMesh is a fictitious application that allows customers to order coffee in any location, at any time. The CoffeeMesh platform consists of a collection of microservices that implement different capabilities, such as processing orders and scheduling deliveries. We’ll undertake a formal analysis and design of the CoffeeMesh platform in chapter 3. To give you a taste of the kinds of things you’ll learn in this book, we’ll begin implementing the API of CoffeeMesh’s orders service in chapter 2. Before we close this chapter, I’d like to dedicate a section to explaining what you’ll learn from this book.

## 1.6 Who this book is for and what you will learn

To make the most out of this book, you should be familiar with the basics of web development. The code examples in the book are in Python, so a basic understanding of Python is beneficial but not necessary to be able to follow along with them. You do not need to have knowledge of web APIs or microservices, as we will explain these technologies in depth. It is useful if you are familiar with the model-view-controller (MVC) pattern for web development or its variants, such as the model-template-view (MTV) pattern implemented by Python’s popular Django framework. We will draw comparisons with these patterns from time to time to illustrate certain concepts. Basic familiarity with Docker and cloud computing will be useful to get through the chapters about deployments, but I’ll do my 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

* Service decomposition strategies for designing microservice architectures
* How to design REST APIs and how to document them using the OpenAPI specification
* How to build REST APIs in Python using popular frameworks like FastAPI and Flask
* How to design and consume GraphQL APIs and how to build them using Python’s Ariadne framework
* How to test your APIs using property-based testing and API testing frameworks such as Dredd and Schemathesis
* Useful design patterns to achieve loose coupling in your microservices
* How to add authentication and authorization to your APIs using Open Authorization (OAuth) and OpenID Connect (OIDC)
* How to deploy your microservices using Docker and Kubernetes to AWS

By the end of this book, you will be familiar with the benefits that microservices architectures bring for web applications as well as the challenges and difficulties that come with them. You will know how to integrate microservices using APIs, you will know how to build and document those APIs using standards and best practices, and you will be prepared to define the domain of an API with clear application boundaries. Finally, you’ll also know how to test, deploy, and secure your microservice APIs.

## Summary

* Microservices are an architectural pattern in which components of a system are designed and built as independently deployed services. This results in smaller and more maintainable code bases and allows services to be optimized and scaled independently of each other.
* Monoliths are an architectural pattern in which whole applications are deployed in a single build and run in the same process. This makes the application easier to deploy and monitor, but it also makes deployments more challenging when the code base grows large.
* Applications can have multiple types of interfaces, such as UIs, CLIs, and APIs. An API is an interface that allows us to interact with an application programmatically from our code or terminal.
* A web API is an API that runs on a web server and uses HTTP for data transport. We use web APIs to expose service capabilities through the internet.
* Microservices talk to each other using smart endpoints and “dumb pipes.” A dumb pipe is a pipe that simply transfers data from one component to another. A great example of a dumb pipe for microservices is HTTP, which exchanges data between the API client and the API server without knowing anything about the API protocol being used. Therefore, web APIs are a great technology for driving integrations between microservices.
* Despite their benefits, microservices also bring the following challenges:
  + *Effective service decomposition*—We must design services with clear boundaries around specific subdomains; otherwise, we risk building a “distributed monolith.”
  + *Microservice integration tests*—Running integration tests for all microservices is challenging, but we can reduce the risk of integration failures by ensuring APIs are correctly implemented.
  + *Handling service unavailability*—Collaborating services are vulnerable to service unavailability, request timeouts, and processing errors, and therefore must be able to handle those scenarios.
  + *Tracing distributed transactions*—Tracing errors across multiple services is challenging and requires software telemetry tools that allow you to centralize logs, enable API visibility, and trace requests across services.
  + *Increased operational complexity and infrastructure overhead*—Each microservice requires its own infrastructure provisioning, including servers, monitoring systems, and alerts, so you need to invest additional efforts in infrastructure automation.
* Documentation-driven development is an API development workflow that works in three stages:
  + Design and document the API.
  + Build the API against the documentation.
  + Test the API against the documentation.

By putting API documentation at the forefront of the development process, documentation-driven development helps you avoid many common problems that API developers face and therefore reduce the chances of API integration failure.

**1** Sam Newman, *Building Microservices* (O’Reilly, 2015), p. 2.

**2** For a comprehensive view of the different interfaces that can be used to enable communication between microservices, see Chris Richardson, *Microservices Patterns* (Manning, 2019).

**3** For a thorough analysis of strategic architectural decisions around monoliths and microservices, see Vernon, Vaughn and Tomasz Jaskula, *Strategic Monoliths and Microservices* (Addison-Wesley, 2021).

**4** For a more comprehensive analysis of the history of microservices architecture and its precursors, see Nicola Dragoni et al, “Microservices: Yesterday, Today, and Tomorrow,” *Present and Ulterior Software Engineering* (Springer, 2017), pp. 195–216.

**5** Allen Wang and Sudhir Tonse, “Announcing Ribbon: Tying the Netflix Mid-Tier Services Together,” *Netflix Technology Blog*, January 18, 2013, <https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62>. For an excellent discussion of the difference between service-oriented architecture (SOA) and microservices architecture, see Richardson, *Microservices Patterns*, pp. 13–14.

**6** Newman, *Building Microservices*, pp. 7–8.

**7** To learn more about API development workflows and how to use API mock servers to build the client, check out my presentation “API Development Workflows for Successful Integrations,” Manning API Conference, August 3, 2021, <https://youtu.be/SUKqmEX_uwg>.

**8** To learn how API server and client developers can leverage API documentation in their software development process, check out my talk “Leveraging API Documentation to Deliver Reliable API Integrations,” API Specifications Conference, September 28–29, 2021, <https://youtu.be/kAWvM-CVcnw>.

# 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
* Testing the API using a Swagger UI

In this chapter, we implement the API for the orders service, which is one of the microservices of the CoffeeMesh website, the project we introduced in section 1.5. CoffeeMesh is an application that makes and delivers coffee on demand at any time, wherever you are. The orders service allows customers to place orders with CoffeeMesh. As we implement the orders API, you will get an early look into the concepts and processes that we dissect in more detail throughout this book. The code for this chapter is available under the ch02 folder of the GitHub repository provided with this book.

## 2.1 Introducing the orders API specification

Let’s begin by analyzing the requirements of the orders API. Using the orders API, we can place orders, update them, retrieve their details, or cancel them. The orders API specification is available in a file named ch02/oas.yaml in the GitHub repository for this book. OAS stands for *OpenAPI specification*, which is a standard format for documenting REST APIs. In chapter 5, you’ll learn to document your APIs using OpenAPI. As you can see in figure 2.1, the API specification describes a REST API with four main URL paths:

* /orders—Allows us to retrieve lists of orders (GET) and create orders (POST).
* /orders/{order\_id}—Allows us to retrieve the details of a specific order (GET), to update an 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).

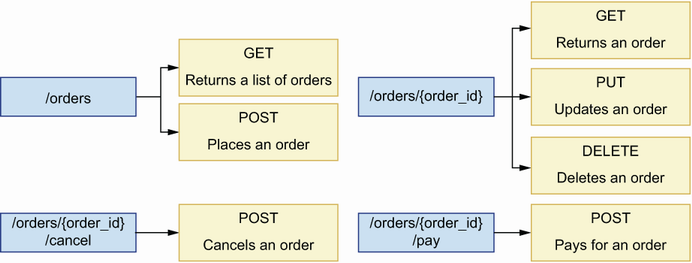


Figure 2.1 The orders API exposes seven endpoints structured around four URL paths. Each endpoint implements different capabilities, such as placing and cancelling an order.

In addition to documenting the API endpoints, the specification also includes data models that tell us what the data exchanged over those endpoints looks like. In OpenAPI, we call those models *schemas*, and you can find them within the components section of the orders API specification. Schemas tell us what properties must be included in a payload and what their types are.

For example, the OrderItemSchema schema 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 enforce the presence of the product and the size properties in the payload before we try to create the order.

Listing 2.1 Specification for OrderItemSchema

# file: oas.yaml

OrderItemSchema:

type: object

required:

- product

- size

properties:

product:

type: string

size:

type: string

enum:

- small

- medium

- big

quantity:

type: integer

default: 1

minimum: 1

Now that we understand the requirements for building the orders API, let’s look at the architectural layout we will use for the implementation.

## 2.2 High-level architecture of the orders application

This section offers a high-level overview of the orders API’s architectural layout. Our goal is to identify the layers of the application and to enforce clear boundaries and separation of concerns between all layers.

As you can see in figure 2.2, we organize into three layers: the API layer, the business layer, and the data layer.

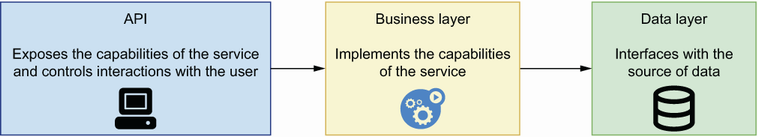


Figure 2.2 To enforce separation of concerns among the different components of our service, we structure our code around three layers: the data layer knows how to interface with the source of data; the business layer implements the service’s capabilities; and the interface layer implements the service’s API.

This way of structuring the application is an adaptation of the three-tier architecture pattern, which structures applications into a data layer, a business layer, and a presentation layer. As you can see in figure 2.3, the data layer is the part of the application that knows how to persist data so that we can retrieve it later. The data layer implements the data models required for interfacing with our source of data. For example, if our persistent storage is an SQL database, the models in the data layer will represent the tables in the database, often with the help of an object relational mapper (ORM) framework.

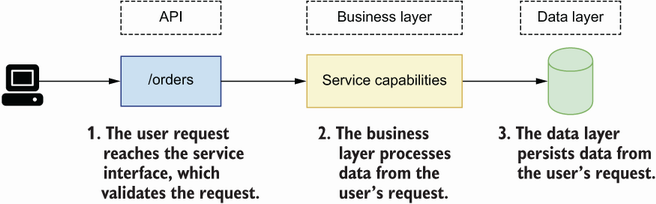


Figure 2.3 When a user request reaches the orders service, it’s first validated by the interface layer. Then the interface layer interfaces with the business layer to process the request. After processing, the data layer persists the data contained in the request.

The business layer implements our service’s capabilities. It controls the interactions between the API layer and the data layer. For the orders service, it’s the part that knows what to do to place, cancel, or pay for an order.

The API layer of a service is different from the business layer. The business layer implements the capabilities of a service, while the API layer is an adapter on top of the application logic that exposes the service’s capabilities to its consumers. Figure 2.2 illustrates this relationship among the layers of a service, while figure 2.3 illustrates how a user request is processed by each layer.

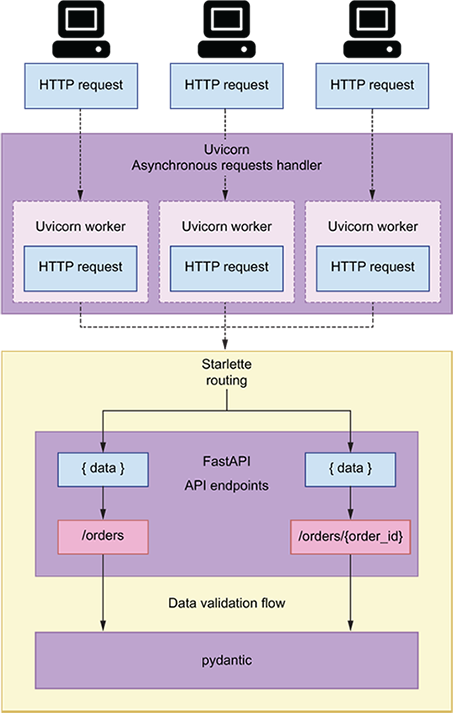
The API layer is an adapter on top of the business layer. Its most important job is validating incoming requests and returning the expected responses. The API layer communicates with the business layer, passing the data sent by the user, so that resources can be processed and persisted in the server. The API layer is equivalent to the presentation layer in three-tier architecture. 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 implementation of the API into progressive steps. In the first step, we produce a minimal implementation of the endpoints with mock responses. In the following sections of this chapter, we enhance the implementation by adding data validation and dynamic responses. You’ll also learn about the FastAPI library and how you can use it to build a web API.

What is FastAPI?

FastAPI (<https://github.com/tiangolo/fastapi>) is a web API framework built on top of Starlette (<https://github.com/encode/starlette>). Starlette is a high-performance, lightweight, asynchronous server gateway interface (ASGI) web framework, which means that we can implement our services as a collection of asynchronous tasks to gain performance in our applications. In addition, FastAPI uses pydantic (<https://github.com/samuelcolvin/pydantic/>) for data validation. The following figure illustrates how all these different technologies fit together.



Uvicorn (<https://github.com/encode/uvicorn>) is an asynchronous web server commonly used to run Starlette applications. Uvicorn handles HTTP requests and passes them on to Starlette, which functions within your application 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.

Before we start implementing the API, we need to set up our environment for this project. Create a folder named ch02 and move into it using the cd command in your terminal. We’ll use Pipenv to install and manage our dependencies.

About dependencies

If you want to make sure you use the same dependencies that I used when writing this book, you can fetch the ch02/Pipfile and ch02/Pipfile.lock files from the GitHub repository for this book and run pipenv install.

Pipfile describes the environment that we wish to create with Pipenv. Among other things, Pipfile contains the version of Python that must be used to create the environment and the URLs of the PyPi repositories that must be used to pull the dependencies. Pipenv also makes it easier to keep production dependencies separate from development dependencies by providing specific installation flags for each set. For example, to install pytest we run pipenv install pytest --dev. Pipenv also exposes 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.

Pipenv is a dependency management tool for Python that guarantees that the 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 Pipfile.lock, which contains a description of the exact package versions that were installed.

Listing 2.2 Creating a virtual environment and installing dependencies with pipenv

$ pipenv --three ①

$ pipenv install fastapi uvicorn ②

$ pipenv shell ③

① Create a virtual environment using pipenv and setting the runtime to Python 3.

② Install FastAPI and Uvicorn.

③ Activate the virtual environment.

Now that our dependencies are installed, let’s build the API. First, copy the API specification under ch02/oas.yaml in the GitHub repository for this book in the ch02 folder we created earlier. Then create a subfolder named orders, which will contain our API implementation. Within the orders folder, create a file called app.py. Create another subfolder called orders/api, and within that folder create a file called orders/api/ api.py. At this point, the project structure should look like this:

.

├── Pipfile

├── Pipfile.lock

├── oas.yaml

└── orders

├── api

│ └── api.py

└── app.py

Listing 2.3 shows how to create an instance of the FastAPI application in file orders/ app.py. The instance of the FastAPI class from FastAPI is an object that represents the API we are implementing. It provides *decorators* (functions that add additional functionality to a function or class) that allow us to register our view functions.[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/02.htm#pgfId-1073824)

Listing 2.3 Creating an instance of the FastAPI application

# file: orders/app.py

from fastapi import FastAPI

app = FastAPI(debug=True) ①

from orders.api import api ②

① We create an instance of the FastAPI class. This object represents our API application.

② We import the api module so that our view functions can be registered at load time.

Listing 2.4 shows a minimal implementation of our API endpoints. The code goes within the orders/api/api.py file. We declare a static order object, and we return the same data in all the endpoints except the DELETE /orders/{order\_id} endpoint, which returns an empty response. Later, we’ll change the implementation to use a dynamic list of orders. FastAPI decorators transform the data we return in every function into an HTTP response; they also map our functions to a specific URL in our server. By default, FastAPI includes 200 (OK) status codes in our responses, but we can override this behavior by using the status\_code parameter in the routes decorators, like we do in the POST /orders and in the DELETE /orders/{order\_id} endpoints.

Listing 2.4 Minimal implementation of the orders API

# file: orders/api/api.py

from datetime import datetime

from uuid import UUID

from starlette.responses import Response

from starlette import status

from orders.app import app

order = { ①

'id': 'ff0f1355-e821-4178-9567-550dec27a373',

'status': "delivered",

'created': datetime.utcnow(),

'order': [

{

'product': 'cappuccino',

'size': 'medium',

'quantity': 1

}

]

}

@app.get('/orders') ②

def get\_orders():

return {'orders': [orders]}

@app.post('/orders', status\_code=status.HTTP\_201\_CREATED) ③

def create\_order():

return order

@app.get('/orders/{order\_id}') ④

def get\_order(order\_id: UUID): ⑤

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) ⑥

@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

① We define an order object to return in our responses.

② We register a GET endpoint for the /orders URL path.

③ We specify that the response’s status code is 201 (Created).

④ We define URL parameters, such as order\_id, within curly brackets.

⑤ We capture the URL parameter as a function argument.

⑥ We use HTTPStatus.NO\_CONTENT.value to return an empty response.

FastAPI exposes decorators named after HTTP methods, such as get() and post(). We use these decorators to register our API endpoints. FastAPI’s decorators take at least one argument, which is the URL path we want to register.

Our view functions can take any number of parameters. If the name of the parameter matches the name of a URL path parameter, FastAPI passes the path parameter from the URL to our view function on invocation. For example, as you can see in figure 2.4, 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 navigates to 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. FastAPI allows us to specify the type and format of the URL path parameter by using type hints. In listing 2.4, we specify that order\_id’s type is a *universally unique identifier* (UUID). FastAPI will invalidate any calls in which order\_id doesn’t follow that format.

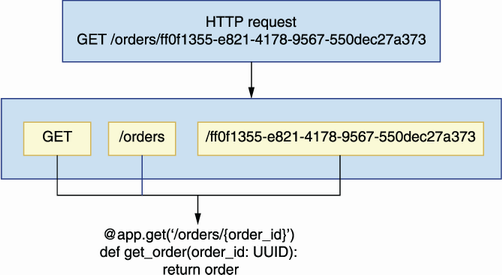


Figure 2.4 FastAPI knows how to map a request to the right function, and it passes any relevant parameters from the request to the function. In this illustration, a GET request on the /orders/{order\_id} endpoint with order\_id set to ff0f1355-e821-4178-9567-550dec27a373 is passed to the get\_order() function.

FastAPI responses include a 200 (OK) status code by default, but we can change this behavior by setting the status\_code parameter in the endpoints’ decorators. In listing 2.4, we set status\_code to 201 (Created) in the POST /orders endpoint, and to 204 (No Content) in the DELETE /orders/{order\_id} endpoint. For a detailed explanation of status codes, see section 4.6 in chapter 4.

You can now run the app to get a feeling of what the API looks like by executing the following command from the top-level orders directory:

$ uvicorn orders.app:app --reload

This command loads the server with hot reloading enabled. *Hot reloading* restarts your server whenever you make changes to your files. Visit the http://127.0.0.1:8000/docs URL in a browser and you will see an interactive display of the API documentation generated by FastAPI from our code (see figure 2.5 for an illustration). This visualization is called Swagger UI, and it’s one of the most popular ways of visualizing REST APIs. Another popular visualization is Redoc, which is also supported by FastAPI under the http://127.0.0.1:8000/redoc URL.

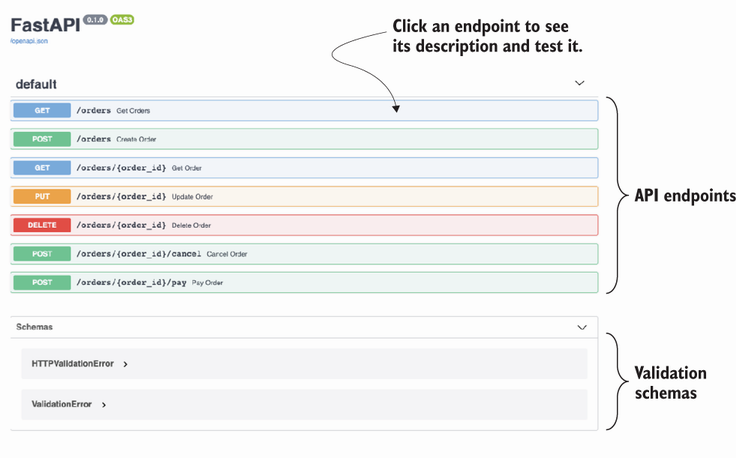


Figure 2.5 View of the Swagger UI dynamically generated by FastAPI from our code. We can use this view to test the implementation of our endpoints.

If you click on any of the endpoints represented in the Swagger UI, you will see additional documentation about the endpoint. You will also see a Try it Out button, which gives you the opportunity to test the endpoint directly from this UI. Click that button, then click Execute, and you will get the hardcoded response we included in our endpoints (see figure 2.6 for an illustration).

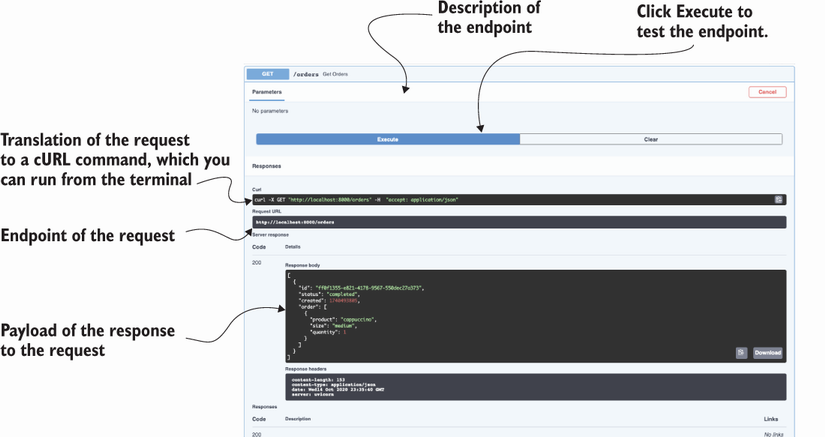


Figure 2.6 To test an endpoint, click it to expand it. You’ll see a Try it Out button on the top-right corner of the endpoint’s description. Click that button, and then click the Execute button. This triggers a request to the server, and you’ll be able to see the response.

Now that we have the basic skeleton of our API, we’ll 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.

## 2.4 Implementing data validation models with pydantic

Now that we have implemented the main layout for the URL paths of our API, we need to add validation for incoming payloads and how we marshal our outgoing responses. Data validation and marshalling are crucial operations in an API, and to deliver a successful API integration, we need to get them right. In the following sections, you’ll learn to add robust data validation and marshalling capabilities to your APIs. FastAPI uses pydantic for data validation, so we’ll start by learning to create pydantic models in this section.

**DEFINITION** *Marshalling* is the process of transforming an in-memory data structure into a format suitable for storage or transmission over a network. In the context of web APIs, marshalling refers to the process of transforming an object into a data structure that can be serialized into a content type of choice, like XML or JSON, with explicit mappings for the object attributes (see figure 2.7 for an illustration).

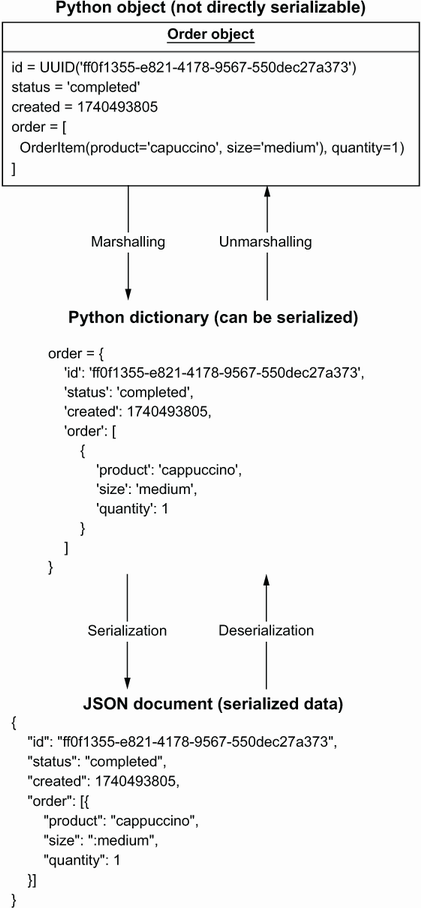


Figure 2.7 To build a response payload from a Python object, we first marshal the object into a serializable data structure, with explicit mapping of attributes between the object and the new structure. Deserializing the payload gives us back an object identical to the one we serialized.

The orders API specification contains three schemas: CreateOrderSchema, GetOrderSchema, and OrderItemSchema. Let’s analyze these schemas to make sure we understand how we need to implement our validation models.

Listing 2.5 Specification for the orders API schemas

# file: oas.yaml

components:

schemas:

OrderItemSchema:

type: object ①

required: ②

- product

- size

properties: ③

product:

type: string

size:

type: string

enum: ④

- small

- medium

- big

quantity:

type: integer

default: 1 ⑤

minimum: 1 ⑥

CreateOrderSchema:

type: object

required:

- order

properties:

order:

type: array

items: ⑦

$ref: '#/components/schemas/OrderItemSchema' ⑧

GetOrderSchema:

type: object

required:

- order

- id

- created

- status

properties:

id:

type: string

format: uuid

created:

type: string

format: date-time

status:

type: string

enum:

- created

- progress

- cancelled

- dispatched

- delivered

order:

type: array

items:

$ref: '#/components/schemas/OrderItemSchema'

① Every schema has a type, which in this case is an object.

② We list compulsory properties under the required keyword.

③ We list object properties under the properties keyword.

④ We constrain the values of a property using an enumeration.

⑤ Attributes can have a default value.

⑥ We can also specify a minimum value for a property.

⑦ We specify the type of the items in the array using the items keyword.

⑧ We use a JSON pointer to reference another schema within the same document.

We use GetOrderSchema when we return the details of an order from the server and CreateOrderSchema to validate an order placed by a customer. Figure 2.8 illustrates how the data validation flow works for CreateOrderSchema. As you can see, 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 that the product and size properties are present in the payload and that they have the right type. Figure 2.8 shows what happens when the quantity property is missing from the payload. In that case, we set the property to its default value of 1 in the server.

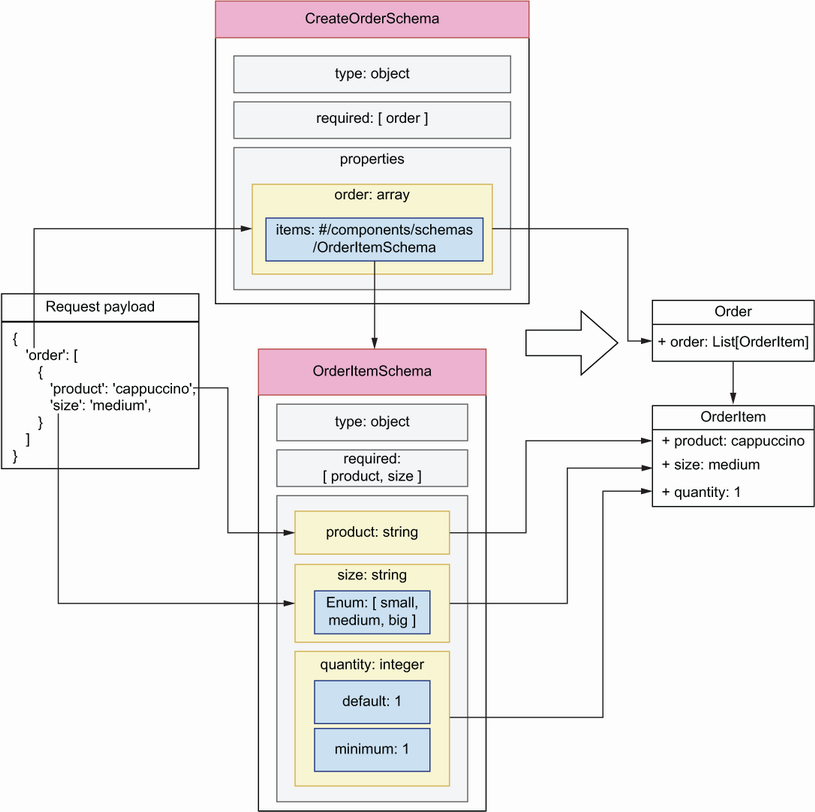


Figure 2.8 Data validation flow for request payloads against the CreateOrderSchema model. The diagram shows how each property of the request payload is validated against the properties defined in the schema and how we build an object from the resulting validation.

Now that we understand our API schemas, it’s time to implement them. Create a new file called orders/api/schemas.py. This file will contain our pydantic models. Listing 2.6 shows how we implement CreateOrderSchema, GetOrderSchema, and OrderItemSchema using pydantic. The code in listing 2.6 goes in the orders/api/schemas.py module. We define every schema as a class that inherits from pydantic’s BaseModel class, and we specify the type of every attribute using Python type hints. For attributes that can only take on a limited selection of values, we define an enumeration class. In this case, we define enumerations for the size and status properties. We set the type of OrderItemSchema’s quantity property to pydantic’s conint type, which enforces integer values. We also specify that quantity is an optional property and that its values should be equal or greater than 1, and we give it a default value of 1. Finally, we use pydantic’s conlist type to define CreateOrderSchema’s order property as a list with at least one element.

Listing 2.6 Implementation of the validation models using pydantic

# file: orders/api/schemas.py

from enum import Enum

from typing import List

from uuid import UUID

from pydantic import BaseModel, Field, conlist, conint

class Size(Enum): ①

small = 'small'

medium = 'medium'

big = 'big'

class Status(Enum):

created = 'created'

progress = 'progress'

cancelled = 'cancelled'

dispatched = 'dispatched'

delivered = 'delivered'

class OrderItemSchema(BaseModel): ②

product: str ③

size: Size ④

quantity: Optional[conint(ge=1, strict=True)] = 1 ⑤

class CreateOrderSchema(BaseModel):

order: conlist(OrderItemSchema, min\_items=1) ⑥

class GetOrderSchema(CreateOrderSchema):

id: UUID

created: datetime

status: Status

class GetOrdersSchema(BaseModel):

orders: List[GetOrderSchema]

① We declare an enumeration schema.

② Every pydantic model inherits from pydantic’s BaseModel.

③ We use Python-type hints to specify the type of an attribute.

④ We constrain the values of a property by setting its type to an enumeration.

⑤ We specify quantity’s minimum value, and we give it a default.

⑥ We use pydantic’s conlist type to define a list with at least one element.

Now that our validation models are implemented, in the following sections we’ll link them with the API to validate and marshal payloads.

## 2.5 Validating request payloads with pydantic

In this section, we use the models we implemented in section 2.4 to validate request payloads. How do we access request payloads within our view functions? We intercept request payloads by declaring them as a parameter of the view function, and to validate them we set their type to the relevant pydantic model.

Listing 2.7 Hooking validation models up with the API endpoints

# file: orders/api/api.py

from uuid import UUID

from starlette.responses import Response

from starlette import status

from orders.app import app

**from orders.api.schemas import CreateOrderSchema**  ①

...

@app.post('/orders', status\_code=status.HTTP\_201\_CREATED)

def create\_order(**order\_details: CreateOrderSchema**): ②

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

...

① We import the pydantic models so that we can use them for validation.

② We intercept a payload by declaring it as a parameter in our function, and we use type hints to validate it.

If you kept the application running, the changes are loaded automatically by the server, so you just need to refresh the browser to update the UI. If you click 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. Now, if you try editing the payload to remove any of the required fields, for example, the product field, and you send it to the server, 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 that 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 syntax that allows you to represent the path to a specific value within a JSON document. If this is the first time you’ve encountered JSON pointers, think of them as a different way of representing dictionary syntax and index notation in Python. For example, the error message "loc: /body/order/0/product" is roughly equivalent to the following notation in Python: loc['body']['order'][0]['product']. Figure 2.9 shows you how to interpret the JSON pointer from the error message to identify the source of the problem in the payload.

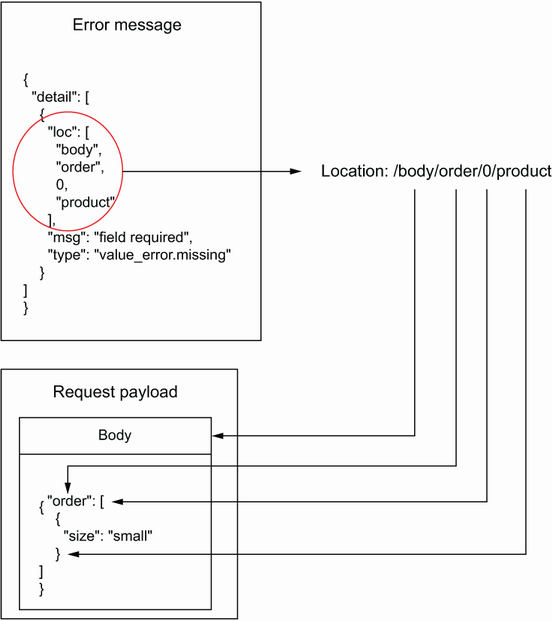


Figure 2.9 When a request fails due a malformed payload, we get a response with an error message. The error message uses a JSON pointer to tell us where the error is. In this case, the error message says that the property /body/order/0/product is missing from the payload.

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

{

"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 make a typo in the payload? For example, imagine a client sent the following payload to the server:

{

"order": [

{

"product": "string",

"size": "small",

"**quantit**": 5

}

]

}

In this case, FastAPI assumes that the quantity property is missing and that the client wishes to set its value to 1. This result could lead to confusion between the client and the server, and in such cases invalidating payloads with illegal properties helps us make the API integration more reliable. In chapter 6, you’ll learn to handle those situations.

One edge case with optional properties, such as OrderItemSchema’s quantity, is that pydantic assumes they’re nullable and therefore will accept payloads with quantity set to null. For example, if we send the following payload to the POST /orders endpoint, our server will accept it:

{

"order": [

{

"product": "string",

"size": "small",

**"quantity": null**

}

]

}

In terms of API integrations, optional isn’t quite the same as nullable: a property can be optional because it has a default value, but that doesn’t mean it can be null. To enforce the right behavior in pydantic, we need to include an additional validation rule that prevents users from setting the value of quantity to null. We use pydantic’s validator() decorator to define additional validation rules for our models.

Listing 2.8 Including additional validation rules for pydantic models

# file: orders/api/schemas.py

from datetime import datetime

from enum import Enum

from typing import List, Optional

from uuid import UUID

from pydantic import BaseModel, conint, **validator**

...

class OrderItemSchema(BaseModel):

product: str

size: Size

quantity: Optional[conint(ge=1, strict=True)] = 1

**@validator('quantity')**

**def quantity\_non\_nullable(cls, value):**

**assert value is not None, 'quantity may not be None'**

**return value**

**...**

Now that we know how to test our API implementation using a Swagger UI, let’s see how we use pydantic to validate and serialize our API responses.

## 2.6 Marshalling and validating response payloads with pydantic

In this section, we’ll use the pydantic models implemented in section 2.4 to marshal and validate the response payloads of our API. Malformed payloads are one of the most common causes of API integration failures, so this step is crucial to deliver a robust API. 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.

**NOTE** Malformed response payloads are a common source of API integration failures. You can avoid this problem by validating your response payloads before they leave the server. In FastAPI, this is easily done by setting the response\_model parameter of a route decorator.

Listing 2.9 shows how we use pydantic models to validate the responses from the GET /orders and the POST /orders endpoints. As you can see, we set the response\_model parameter to a pydantic model in FastAPI’s route decorators. We follow the same approach to validate responses from all the other endpoints except the DELETE /orders/{order\_id} endpoint, which returns an empty response. Feel free to check out the code in the GitHub repository for this book for the full implementation.

Listing 2.9 Hooking validation models for responses in the API endpoints

# file: orders/api/api.py

from uuid import UUID

from starlette.responses import Response

from starlette import status

from orders.app import app

from orders.api.schemas import (

**GetOrderSchema,**

CreateOrderSchema,

**GetOrdersSchema,**

)

...

@app.get('/orders', **response\_model=GetOrdersSchema**)

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

Now that we have response models, FastAPI will raise an error if a required property is missing from a response payload. It will also remove any properties that are not part of the schema, and it will try to cast each property into the right type. Let’s see this behavior at work.

In a browser, visit the http://127.0.0.1:8000/docs URL to load the Swagger UI for our API. Then head over to the GET /orders endpoint and send a request. You’ll get the order that we hardcoded at the top of the orders/api/api.py file. Let’s make some modifications to that payload to see how FastAPI handles them. To begin, let’s add an additional property called updated:

**# orders/api/api.py**

**...**

order = {

'id': 'ff0f1355-e821-4178-9567-550dec27a373',

'status': 'delivered',

'created': datetime.utcnow(),

**'updated': datetime.utcnow(),**

'order': [

{

'product': 'cappuccino',

'size': 'medium',

'quantity': 1

}

]

}

...

If we call the GET /orders endpoint again, we’ll get the same response we obtained before, without the updated property since it isn’t part of the GetOrderSchema model:

[

{

"order": [

{

"product": "cappuccino",

"size": "medium",

"quantity": 1

}

],

"id": "ff0f1355-e821-4178-9567-550dec27a373",

"created": datetime.utcnow(),

"status": "delivered"

}

]

Let’s now remove the created property from the order payload and call the GET /orders endpoint again:

**# orders/api/api.py**

**...**

order = {

'id': 'ff0f1355-e821-4178-9567-550dec27a373',

'status': "delivered",

'updated': datetime.utcnow(),

'order': [

{

'product': 'cappuccino',

'size': 'medium',

'quantity': 1

}

]

}

This time, 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 random string and run another request against the GET /orders endpoint:

**# orders/api/api.py**

**...**

order = {

'id': 'ff0f1355-e821-4178-9567-550dec27a373',

'status': "delivered",

**'created': 'asdf',**

'updated': 1740493905,

'order': [

{

'product': 'cappuccino',

'size': 'medium',

'quantity': 1

}

]

}

...

In this case, FastAPI raises 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)

Our responses are being correctly validated and marshalled. Let’s now add a simple state management mechanism for the application so that we can place orders and change their state through the API.

## 2.7 Adding an in-memory list of orders to the API

So far, our API implementation has returned the same response object. Let’s change that by adding a simple in-memory collection of orders to manage the state of the application. To keep the implementation simple, we’ll represent the collection of orders as a Python list. We’ll manage the list within the view functions of the API layer. In chapter 7, you’ll learn useful patterns to add a robust controller and data persistence layers to the application.

Listing 2.10 shows the changes required for the view functions under api.py to manage the in-memory list of orders in our view functions. The changes in listing 2.9 go into the orders/api/api.py file. We represent the collection of orders as a Python list, and we assign it to the variable ORDERS. To keep it simple, we store the details of every order as a dictionary, and we update them by changing their properties in the dictionary.

Listing 2.10 Managing the application’s state with an in-memory list

# file: orders/api/api.py

**import time**

**import uuid**

**from datetime import datetime**

from uuid import UUID

**from fastapi import HTTPException**

from starlette.responses import Response

from starlette import status

from orders.app import app

from orders.api.schemas import GetOrderSchema, CreateOrderSchema

ORDERS = [] ①

@app.get('/orders', response\_model=**GetOrdersSchema**)

def get\_orders():

**return ORDERS**  ②

@app.post(

'/orders',

status\_code=status.HTTP\_201\_CREATED,

response\_model=GetOrderSchema,

)

def create\_order(order\_details: CreateOrderSchema):

**order = order\_details.dict()**  ③

**order['id'] = uuid.uuid4()**  ④

**order['created'] = datetime.utcnow()**

**order['status'] = 'created'**

**ORDERS.append(order)**  ⑤

**return order**  ⑥

@app.get('/orders/{order\_id}', response\_model=GetOrderSchema)

def get\_order(order\_id: UUID):

**for order in ORDERS:**  ⑦

**if order['id'] == order\_id:**

**return order**

**raise HTTPException(**  ⑧

**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}',

status\_code=status.HTTP\_204\_NO\_CONTENT,

response\_class=Response,

)

def delete\_order(order\_id: UUID):

**for index, order in enumerate(ORDERS):**  ⑨

**if order['id'] == order\_id:**

**ORDERS.pop(index)**

**return 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'**

**)**

① We represent our in-memory list of orders as a Python list.

② To return the list of orders, we simply return the ORDERS list.

③ We transform every order into a dictionary.

④ We enrich the order object with server-side attributes, such as the ID.

⑤ To create the order, we add it to the list.

⑥ After appending the order to the list, we return it.

⑦ To find an order by ID, we iterate the ORDERS list and check their IDs.

⑧ If an order isn’t found, we raise an HTTPException with status\_code set to 404 to return a 404 response.

⑨ We 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 update them by hitting the PUT /orders/{order\_id} endpoint. 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.

We are now able to use the orders API to create orders, update them, pay for them, cancel them, and get their details. You have implemented a fully working web API for a microservice application! You’ve become familiar with a bunch of new libraries to build web APIs, and you’ve seen how to add robust data validation to your APIs. You’ve also learned to put it all together and run it with success. Hopefully, this chapter has sparked your interest and excitement about designing and building microservices exposing web APIs. In the coming chapters, we’ll delve deeper into these topics, and you’ll learn to build and deliver robust and secure microservice API integrations.

## Summary

* To structure microservices into modular layers, we use an adaptation of the three-tier architecture pattern:
  + A data layer that knows how to interface with the source of data
  + A business layer that implements the capabilities of the service
  + An interface or presentation layer that exposes the capabilities of the service through an API
* FastAPI is a popular framework for building web APIs. It’s highly performant, and it has a rich ecosystem of libraries that make it easier to build APIs.
* FastAPI uses pydantic, a popular data validation library for Python. Pydantic uses type hints to create validation rules, which results in clean and easy-to-understand models.
* FastAPI generates a Swagger UI dynamically from our code. A Swagger UI is a popular interactive visualization UI for APIs. Using the Swagger UI, we can easily test if our implementation is correct.

# 3 Designing microservices

This chapter covers

* Principles of microservices design
* Service decomposition by business capability
* Service decomposition by subdomain

When we design a microservices platform, the first questions we face are, “How do you break down a system into microservices? How do you decide where a service ends and another one starts?” In other words, how do you define the boundaries between microservices? In this chapter, you’ll learn to answer these questions and how to evaluate the quality of a microservices architecture by applying a set of design principles.

The process of breaking down a system into microservices is called *service decomposition*. Service decomposition is a fundamental step in the design of our microservices 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 a distributed monolith. In this chapter, you’ll learn two service decomposition strategies: decomposition by business capability and decomposition by subdomains. We’ll see how these methods work and use a practical example to learn to apply them. Before we delve into service decomposition strategies, we introduce the project that will guide our examples throughout this chapter and the rest of the book: CoffeeMesh.

## 3.1 Introducing CoffeeMesh

CoffeeMesh is a fictitious company that allows customers 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 or when they place their order. The production factories owned by CoffeeMesh form a dense network, a mesh of coffee production units that spans several countries. Coffee production is fully automated, and deliveries are carried out by an unmanned fleet of drones operating 24/7.

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 number of pending orders the factory is taking care of, and distance to the customer. Once the items are produced, they’re immediately dispatched to the customer. It’s part of CoffeeMesh’s mission statement that the customer receives each item fresh and hot.

Now that we have an example to work with, let’s see how we design the microservices architecture for the CoffeeMesh platform. Before we learn to apply service decomposition strategies for microservices, the next section teaches you three principles that will guide our designs.

## 3.2 Microservices design principles

What makes a well-designed microservice? As we established in chapter 1, microservices are designed around well-defined business subdomains, they have clearly defined application boundaries, and they communicate with each other through lightweight protocols. What does this mean in practice? In this section, we explore three design principles that help us test whether our microservices are correctly designed:

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

Following these principles will help you avoid the risk of building a distributed monolith. In the following sections, we evaluate our architectural design against these principles, and they help us spot errors in the design.

### 3.2.1 Database-per-service principle

The database-per-service principle states that each microservice owns a specific set of the data, and no other service should have access to such data except through an API. Despite this pattern’s name, it does not mean that each microservice should be connected to a completely different database. It could be different tables in an SQL database or different collections in a NoSQL database. The point of this pattern is to ensure that the data owned by a specific service is not accessed directly by another service.

Figure 3.1 shows how microservices share their data. In the illustration, the orders service calculates the price of a customer order. To calculate the price, the orders service needs the price of each item in the order, which is available in the Products database. It also needs to know whether the user has an applicable discount, which can be checked in the Users database. However, instead of accessing both databases directly, the orders service requests this data from the products and users services.

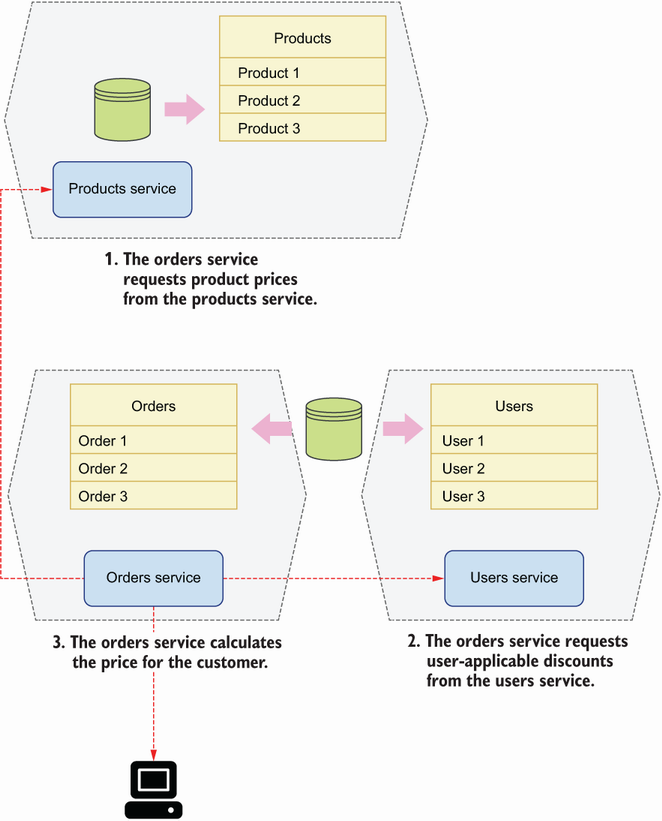


Figure 3.1 Each microservice has its own database, and access to another service’s data happens through an API.

Why is this principle important? Encapsulating data access behind a service allows us to design our data models for optimal access for the service. It also allows us to make changes to the database without breaking another service’s code. If the orders service in figure 3.1 had direct access to the Products database, schema changes in that database would require updates to both the products and orders services. We’d be coupling the orders service’s code to the Products database, and therefore we’d be breaking the loose coupling principle, which we discuss in the next section.

### 3.2.2 Loose coupling principle

*Loose* *coupling* states that we must design services with clear separation of concerns. Loosely coupled services don’t rely on another’s implementation details. What does this mean in practice? This principle has two practical implications:

* Each service can work independently of others. If we have a service that can’t fulfill a single request without calling another service, there’s no clear separation of concerns between both services and they belong together.
* Each service can be updated without impacting other services. If changes to a service require updates to other services, we have tight coupling between those services, and therefore they need to be redesigned.

Figure 3.2 shows a sales forecast service that knows how to calculate a forecast based on historical data. It also shows a historical data service that owns historical sales data. To calculate a forecast, the sales forecast service makes an API call to the historical data service to obtain historical data. In this case, the sales forecast service can’t serve any request without calling the historical data service, and therefore there’s tight coupling between both services. The solution is to redesign both services so that they don’t rely on each other, or to merge them into a single service.

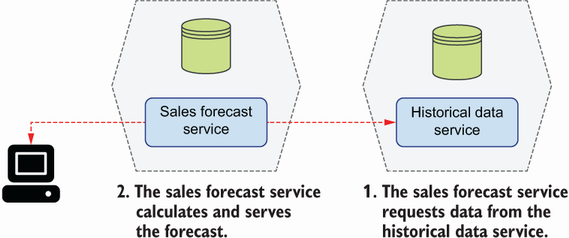


Figure 3.2 When a service can’t serve a single request without calling another service, we say both are tightly coupled.

### 3.2.3 Single Responsibility Principle

The SRP states that we must design components with few responsibilities, and ideally with only one responsibility. When applied to the microservices architecture design, this means we should strive for the design of services around a single business capability or subdomain. In the following sections, you’ll learn how to decompose services by business capability and by subdomain. If you follow any of those methods, you’ll be able to design microservices that follow the SRP.

## 3.3 Service decomposition by business capability

When using decomposition by business capability, we look into the activities 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 build a customer management service; if the business has a claims management team, we build a claims management service; for a kitchen team, we build the corresponding kitchen service; and so on. For businesses that 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 turtle food, and so on. In this scenario, we build microservices for each of these teams.

As you can see in figure 3.3, decomposition by business capability generally results in an architecture that maps every business team to a microservice. Let’s see how we apply this approach to the CoffeeMesh platform.

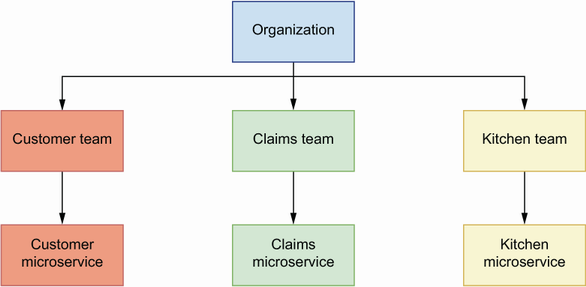


Figure 3.3 Using service decomposition by business capability, we reflect the structure of the business in our microservices architecture.

### 3.3.1 Analyzing the business structure of CoffeeMesh

To apply decomposition by business capability, we need to analyze the structure and organization of the business. Let’s do this analysis for CoffeeMesh. Through the CoffeeMesh website, customers can order different types of coffee-related products out of a catalogue managed by the products team, who is in charge of creating new products. The availability of products and ingredients depends on the CoffeeMesh stock of ingredients at the time of the order, which is looked after by the inventory team.

A sales team is dedicated to improving the experience of ordering products through the CoffeeMesh website. Their goal is to maximize sales and ensure 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 financial infrastructure required to process customer payments and return their money when they cancel an order.

Once a user places an order, the kitchen picks up its details to commence production. Kitchen work is fully automated, and a dedicated team of engineers and chefs called the kitchen team monitors kitchen operations to ensure no faults happen during production. When the order is ready for delivery, a drone picks it up and flies it to the customer. A dedicated team of engineers called the delivery team monitors this process to ensure the operational excellence of the delivery process.

This completes our analysis of the organizational structure of CoffeeMesh. We’re now ready to design a microservices architecture based on this analysis.

### 3.3.2 Decomposing microservices by business capabilities

To decompose services by business capability, we map each business team to a microservice. Based on the analysis in section 3.3.1, we can map the following business teams to microservices:

* *Products team maps to the products service*—This service owns CoffeeMesh product catalogue data. The products team uses this service to maintain CoffeeMesh’s catalogue by adding new products or updating existing products through the service’s interface.
* *Ingredients team maps to the ingredients service*—This service owns data about CoffeeMesh stock of ingredients. The ingredients team uses this service to keep the ingredients database in sync with CoffeeMesh warehouses.
* *Sales team maps to the sales service*—This service guides customers through their journey to place orders and keep track of them. The sales team owns data about customer orders, and it manages the life cycle of each order. It collects data from this service to analyze and improve the customer journey.
* *Finance team maps to the finance service*—This service implements payment processors, and it owns data about user payment details and payment history. The finance team uses this service to keep the company accounts up to date and to ensure payments work correctly.
* *Kitchen team maps to the kitchen service*—This service sends orders to the automated kitchen system and keeps track of its progress. It owns data about the orders produced in the kitchen. The kitchen team collects data from this service to monitor the performance of the automated kitchen system.
* *Delivery team maps to the delivery service*—This service arranges the delivery of the order to the customer 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 that destination. It owns data about every delivery made by CoffeeMesh. The delivery team collects data from this service to monitor the performance of the automated delivery system.

In this microservices architecture, we named every service after the business structure it represents. We did this for convenience in this example, but it does not have to be that way. For example, the finance service could be renamed to payments service, since all user interactions with this service will be related to their payments.

Decomposition by business capability gives us an architecture in which every service maps to a business team. Is this result in agreement with the principles of microservices design we learned in section 3.2? Let’s look at this question.

From the previous analysis, it’s clear that every service owns its own data: the products service owns product data, the ingredients service owns ingredients data, and so on. The SRP also applies, as every service is restricted to one business area: the finance service only processes payments, the delivery service only manages deliveries, and so on.

However, as you can see in figure 3.4, this solution doesn’t satisfy the loose coupling principle. To serve the CoffeeMesh catalogue, the products service needs to determine the availability of each product, which depends on the available stock of ingredients. Since the stock of ingredients data is owned by the ingredients service, the products service needs to make an API call per product to the ingredients service.

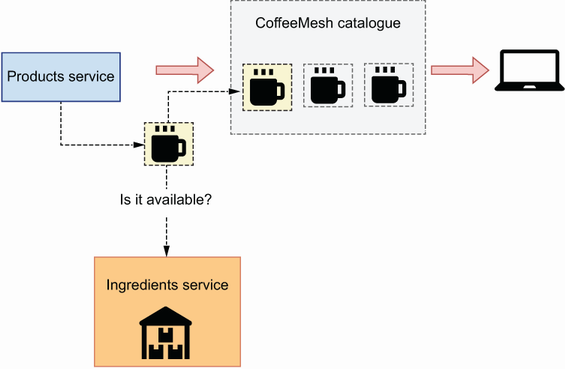


Figure 3.4 To determine whether a product is available, the products service checks the stock of ingredients with the ingredients service.

There’s a high degree of coupling between the products and ingredients services, and therefore both business capabilities should be implemented within the same service. Figure 3.5 shows the final layout of the CoffeeMesh microservices architecture using the decomposition by business capability strategy.

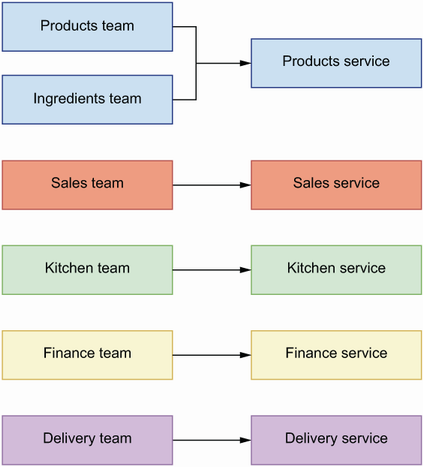


Figure 3.5 When we decompose services by business capability, we map every team to a service.

Now that we know how to decompose services by business capability, let’s see how decomposition by subdomain works.

## 3.4 Service decomposition by subdomains

Decomposition by subdomains is an approach that draws inspiration from the field of *domain-driven design* (DDD)—an approach to software development that focuses on modeling the processes and flows of the business with software using the same language business users employ. When applied to the design of a microservices platform, DDD helps us define the core responsibilities of each service and their boundaries.

### 3.4.1 What is domain-driven design?

DDD is an approach to software that focuses on modeling the processes and flows of the business users. The methods of DDD were best described by Eric Evans in his influential 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. To achieve this alignment, 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 is called *ubiquitous language*.

To create an ubiquitous language, we must identify the core domain of a business, which corresponds with the main activity an organization performs to generate value. For a logistics company, it may be the shipment of products around the world. For an e-commerce company, it may be the sale of products. For a social media platform, it may be feeding a user with relevant content. For a dating app, it may be matching users. For CoffeeMesh, the core domain is to deliver high-quality coffee to customers as quickly as possible regardless of their location.

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 that is not directly related to value generation, but it is fundamental to support it. For a logistics company, it may be providing customer support to the users shipping their products, leasing equipment, managing partnerships with other businesses, and so on. For an e-commerce company, it may be marketing, customer support, warehousing, and so on.

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

### 3.4.2 Applying strategic analysis to CoffeeMesh

How does DDD work in practice? How do we apply it to decompose CoffeeMesh into subdomains? To break down a system into subdomains, it helps to think about the operations the system has to perform to accomplish its goal. With CoffeeMesh, we want to model the process of taking an order and delivering it to the customer. As you can see in figure 3.6, we break down this process into eight steps:

1. When the customer lands on the website, we show them the product catalogue. Each product is marked as available or unavailable. The customer can filter the list by availability and sort it by price (from lowest to highest and highest to lowest).
2. The customer selects products.
3. The customer pays for their order.
4. Once the customer has paid, we pass on the details of the order to the kitchen.
5. The kitchen picks up the order and produces it.
6. The customer monitors progress on their order.
7. Once the order is ready, we arrange its delivery.
8. The customer tracks the drone’s itinerary until their order is delivered.

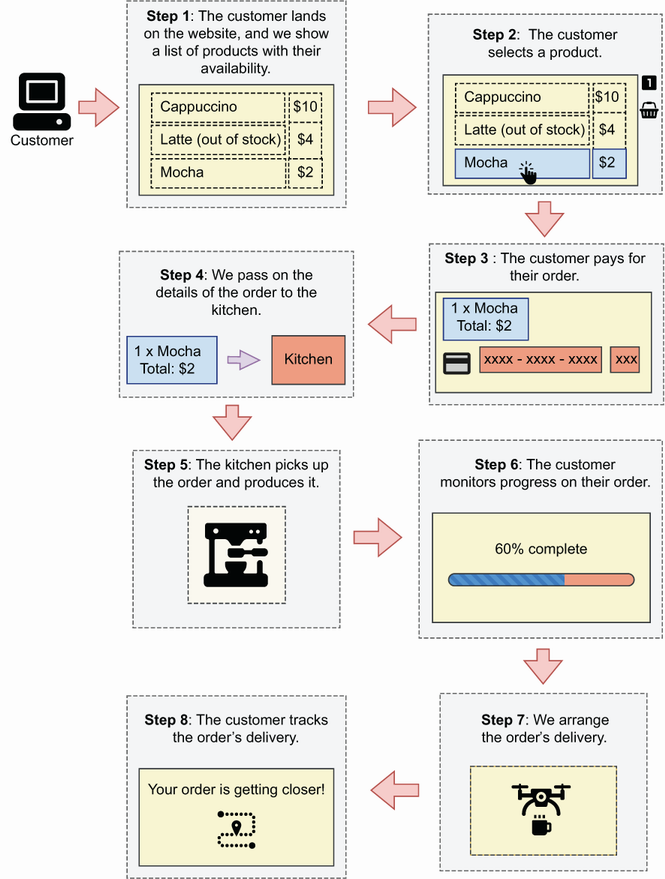


Figure 3.6 To place an order, the customer lands on the CoffeeMesh website, selects items from the product catalogue, and pays for the order. After payment, we pass the order’s details to the kitchen, which produces it while the customer monitors its progress. Finally, we arrange the order’s delivery.

Let’s map each step to its corresponding subdomain (see figure 3.7 for a representation of this analysis). The first step represents a subdomain that serves the CoffeeMesh product catalogue. We can call it the *products subdomain*. This subdomain tells us which products are available and which are not. To do so, the products subdomain tracks the amount of each product and ingredient in stock.

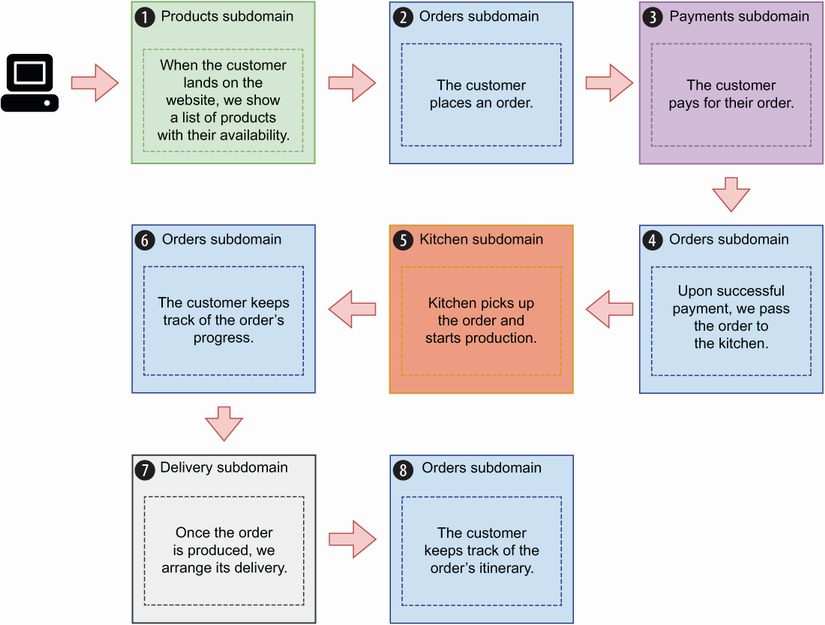


Figure 3.7 We map to a subdomain every step in the process of placing and delivering an order. For example, the process of serving the product catalogue is satisfied by the products subdomain, while the process of taking an order is satisfied by the orders subdomain.

The second step represents a subdomain that allows users to select products. This subdomain manages the life cycle of each order, and we call it the *orders subdomain*. This subdomain owns data about users’ orders, and it exposes an interface that allows us to manage orders and check their status. It hides the complexity of the platform so that the user doesn’t have to know about different endpoints and know what to do with them. The orders subdomain also takes care of the second part of the fourth step: passing the details of the order to the kitchen once the payment has been successfully processed. It also meets the requirements for step 6: allow the user to check the state of their order. As an orders manager, the orders subdomain also works with the delivery subdomain to arrange the delivery.

The third step represents a subdomain that can handle user payments. We will call it the *payments subdomain*. This domain contains specialized logic for payment processing, including card validation, integration with third-party payment providers, handling different methods of payment, and so on. The payments subdomain owns data related to user payments.

The fifth step represents a subdomain that works with the kitchen to manage the production of customer orders. We call it the *kitchen subdomain*. The production system in the kitchen is fully automated, and the kitchen subdomain interfaces with the kitchen system to schedule the production of customer orders and track their progress. Once an order is produced, the kitchen subdomain notifies the orders subdomain, which then arranges its delivery. The kitchen subdomain owns data related to the production of customer orders, and it exposes an interface that allows us to send orders to the kitchen and keep track of their progress. The orders subdomain interfaces with the kitchen subdomain to update the order’s status to meet the requirements for the sixth step.

The seventh step represents a subdomain that interfaces with the automated delivery system. We call it the *delivery subdomain*. This subdomain contains specialized logic to resolve the geolocation of a customer and to calculate the most optimal route to reach them. It manages the fleet of delivery drones and optimizes the deliveries, and it owns data related to all the deliveries. The orders subdomain interfaces with the delivery subdomain to update the itinerary of the customer’s order to meet the requirements for the eighth step.

Using strategic analysis, we obtain a decomposition for CoffeeMesh in five subdomains, which can be mapped to microservices, as each encapsulates a well-defined and clearly differentiated area of logic that owns its own data. DDD’s strategic analysis results in microservices that satisfy the design principles we enumerated in section 3.2: all these subdomains can perform their core tasks without relying on other microservices, and therefore we say they’re loosely coupled; each service owns its own data, hence complying with the database-per-service principle; finally, each service performs tasks within a narrowly defined subdomain, which complies with the SRP.

As you can see in figure 3.8, strategic analysis gives us the following microservices architecture:

* *Products subdomain maps to the products service*—Manages CoffeeMesh’s product catalogue
* *Orders subdomain maps to the orders service*—Manages customer orders
* *Payments subdomain maps to the payments service*—Manages customer payments
* *Kitchen subdomain maps to the kitchen service*—Manages the production of orders in the kitchen
* *Delivery subdomain maps to the delivery service*—Manages customer deliveries

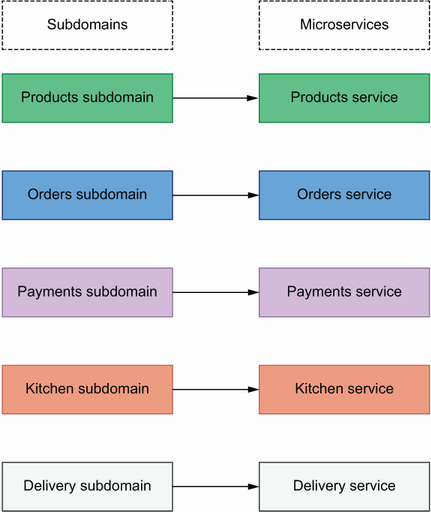


Figure 3.8 Applying DDD’s strategic analysis breaks down the CoffeeMesh platform into five subdomains that can be mapped directly to microservices.

In the next section, we compare the results of DDD’s strategic analysis with the outcome of service decomposition by business capability, and we evaluate the benefits and challenges of each approach.

## 3.5 Decomposition by business capability vs. decomposition by subdomain

Which service decomposition strategy should we use to design our microservices: decomposition by business capability or decomposition by subdomains? While decomposition by business capability focuses on business structure and organization, decomposition by subdomain analyzes business processes and flows. Therefore, both approaches give us different perspectives on the business, and if you can spare the time, the best strategy is to apply both approaches to service decomposition.

Sometimes we can combine the results of both approaches. For example, the CoffeeMesh platform could allow customers to write reviews for each product, and CoffeeMesh could leverage this information to recommend new products to other customers. The company could have an entire team dedicated to this aspect of the business. From a technical point of view, reviews could be just another table in the Products database. However, to facilitate collaboration with the business, it could make sense to build a reviews service. The reviews service would be able to feed new reviews into the recommendation system, and the orders service would use the reviews service’s interface to serve recommendations to new users.

The advantage of decomposition by business capability is that the architecture of the platform aligns 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 can be outdated and reflect old business processes. In that case, the inefficiencies of the business will be mirrored in the microservices architecture. Decomposition by business capability also risks falling out of alignment with the business if the organization is restructured.

When we applied decomposition by business capability in section 3.3.2, we obtained an undesirable division between the products and ingredients services. After further analysis of the dependencies between both services, we concluded that both capabilities should go into the same service. However, in real-life situations, this additional analysis is often missing, and the resulting architecture isn’t optimal. From the analysis in sections 3.3 and 3.4, we can say that decomposition by subdomain gives you a better architectural fit to model the business processes and flows, and if you must choose only one approach, decomposition by subdomain is the better strategy.

Now that we know how to design our microservices, it’s time to design and build their interfaces. In the upcoming chapters, you’ll learn to build REST and GraphQL interfaces for microservices.

## Summary

* We call the process of breaking down a system into microservices service decomposition. Service decomposition defines the boundaries between services, and we must get this process right to avoid the risk of building a distributed monolith.
* Decomposition by business capability analyzes the structure of the business and designs microservices for each team in the organization. This approach aligns the business with our system architecture, but it also reproduces the inefficiencies of the business into the platform.
* Decomposition by subdomains applies DDD to model the processes and flows of the business through subdomains. By using this approach, we design a microservice for each subdomain, which results in a more robust technical design.
* To assess the quality of our microservices architecture, we apply three design principles:
  + *Database-per-service principle*—Each microservice owns its own data, and access to that data happens through the service’s API.
  + *Loose coupling principle*—You must be able to update a service without impacting other services, and each service should be able to work without constantly calling other services.
  + *Single Responsibility Principle*—We must design each service around a specific business capability or subdomain.