Appendix A. Architectural approaches

In this appendix, we discuss a few architectural concepts you encounter. To fully understand everything we discuss in this book, you need to at least be aware and have a high-level overview of these concepts. I'll take you through the concepts of monolith, service-oriented architecture, and microservices. I'll also refer you to other resources you can use to learn the subjects further.

These subjects are complex; many books and dozens of presentations have been delivered on these topics, so I can't say I'll make you an expert in just a few pages, but reading this will help you understand why you use Spring in specific scenarios we discuss in the book. We'll take an app scenario as an example and discuss the change in architectural approaches from the early ages of software development to today.

A.1 The monolithic approach

In this section, we discuss what a monolith is. You'll understand why in the early days developers designed apps monolithically, and then, in the next sections, this will help you understand why other architectural styles appeared.

When developers refer to an app as being "monolithic" or "a monolith," it means that it consists of just one component you deploy and execute. This component implements all its functionalities. For example, consider an app for managing a bookstore. The users manage the products the shop sells, the invoices, the deliveries, and the customers. In figure A.1, the presented system is a monolith, because all these functionalities are part of the same process.

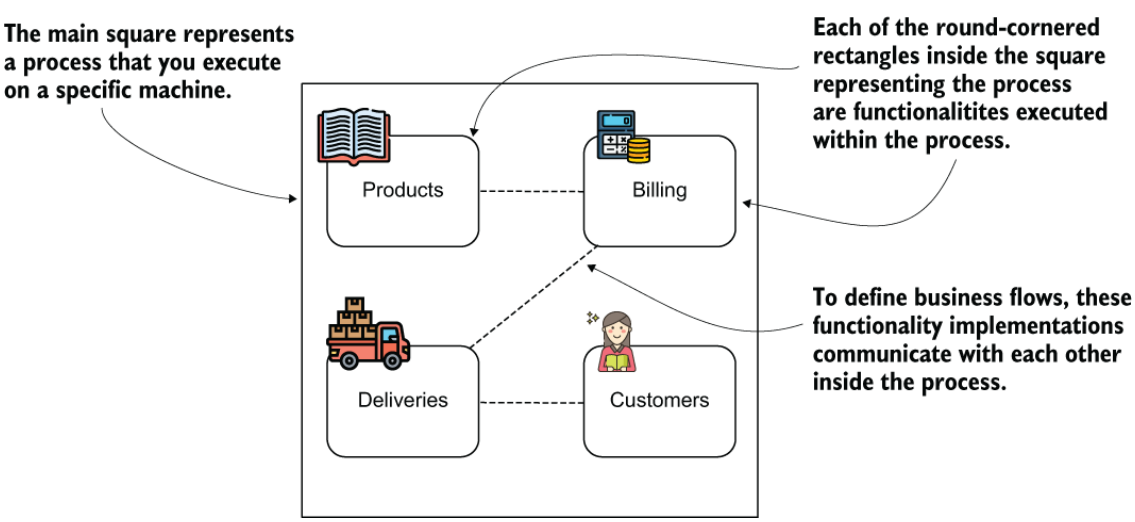

Figure A.1 A monolithic application. The application implements all the functionalities in just one process. The implementations interact with one another inside the process to develop the business flow.

NOTE A business flow is something the user expects to do in the application. For example, when the shop owner sells books, the flow could be as follows: The products functionality reserves some books from the stock, the billing functionality creates an invoice for those books, and the deliveries plan when to deliver the books and notifies the customers. Figure A.2 presents a visual representation of the "sell books" business flow.

NOTE In figure A.2, I simplified the communication between the components to allow you to focus on the discussed subject. The class design, which results in how the components communicate with one another, may be different.

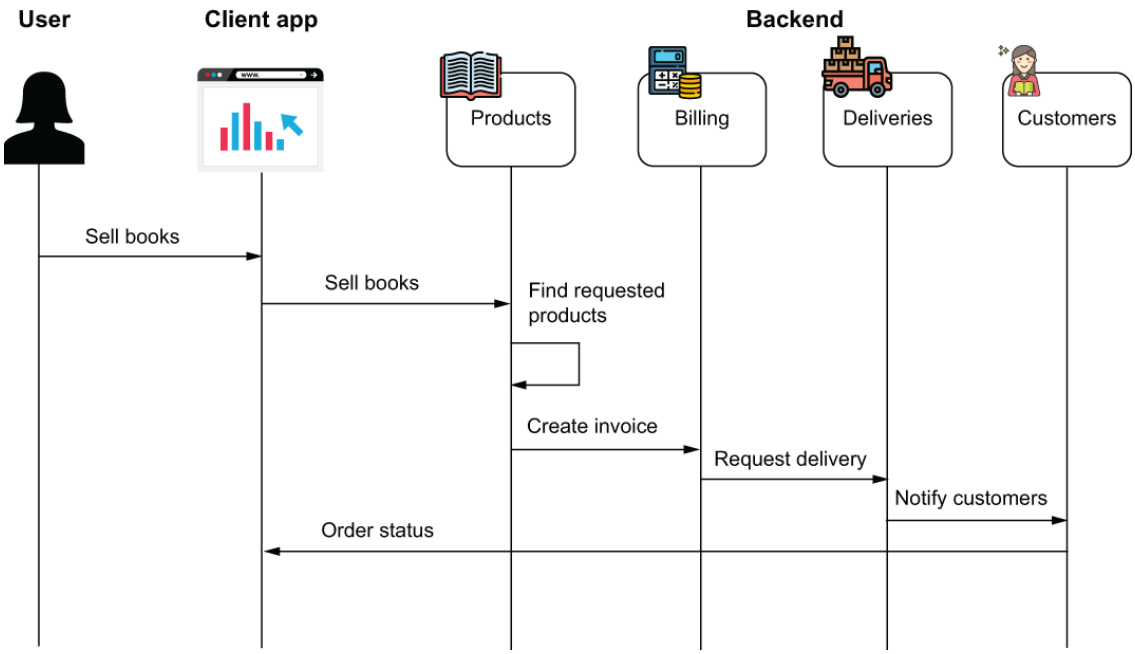

Figure A.2 An example of business flow. The user wants to sell books. The client app sends the request to the backend. Each responsibility of the backend has a role in the entire flow. The functionalities communicate with each other to complete the business flow. In the end, the client app receives the status of the order

Initially, all applications were developed in a monolithic fashion, and this approach worked great in the early days of application development. In the 1990s, the internet was only a network of a few computers, but in a few years, it became a network of billions of devices. Today, technology is no longer for techies; it's for everybody. And this change implied a significant growth in the number of users and data processed for many systems. Thirty years ago, being able to call a cab anywhere you are or even sending a message from the street while waiting to cross the road wasn't something we could have imagined possible.

To deal with this change in the number of users and the growth of data, the apps needed more resources, and using only one process makes the management of the resources more difficult. The number of users and the quantity of data aren't the only things that changed with time; people started to use apps for almost anything they wanted to do remotely. For example, today you can manage your bank accounts while drinking a cappuccino at your favorite café. While this seems easy, it implies more security risks. The systems offering you these services need to be well-secured and reliable.

Of course, all these changes also brought changes in the way the apps are created and developed. Let's consider just the increase in the number of users to simplify our discussion. What's one thing you can do to enable your app to serve more requests? Well, we could run the same app on multiple systems. This way, several instances of running applications will split the requests among them so that the system can deal with a more significant load (figure A.3.). We name this approach *horizontal scaling*. Assuming a linear growth for simplicity; if one running app instance was able to deal with 50,000 simultaneous requests, three running app instances should be able to respond to 150,000 concurrent requests.

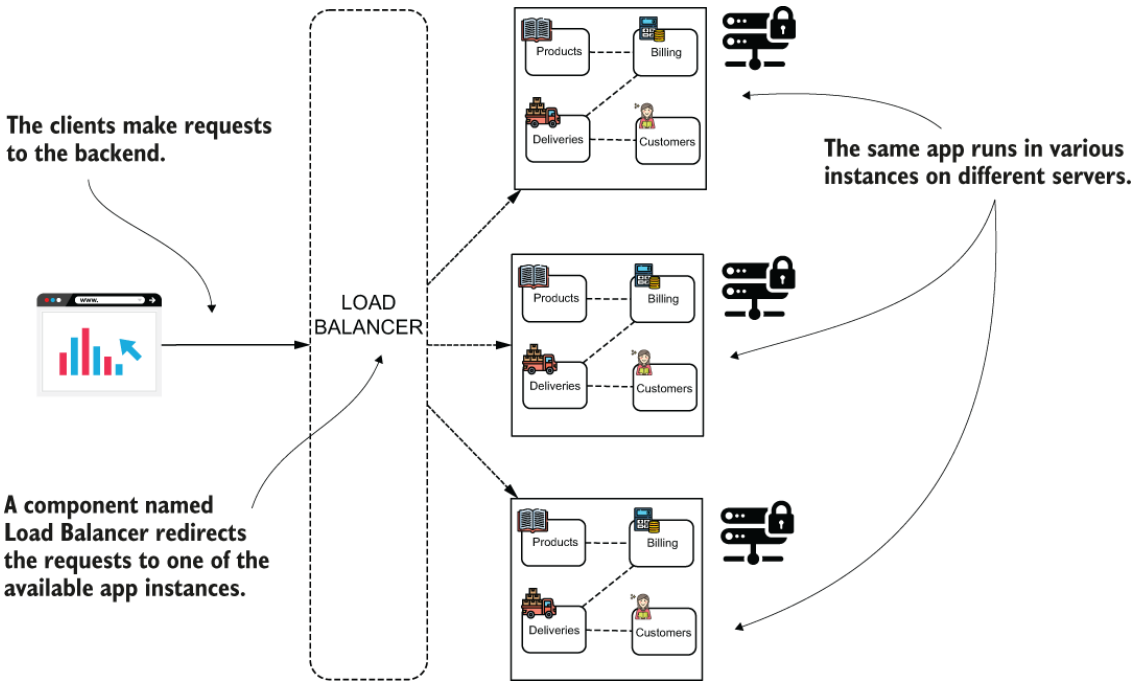

Figure A.3 Horizontal scaling. Running the same instance multiple times enables us to use more resources and serve more client requests.

One other aspect we take into consideration is that, in general, an app is continuously evolving. When you make even a small change in a monolithic app, you need to redeploy everything. At the same time, with a microservices architecture, you benefit from redeploying the service only where you made the change. This simplification is a benefit for the system as well.

Is there a problem with continuing to use a monolithic approach to design an app? There might not be a problem at all. Like with any other technology or technique, designing your app as a monolith could be the best approach for your scenario. We discuss cases in which a monolith isn't the right choice, but I don't want you to get the impression that using

a monolithic architecture is wrong or that the approaches I present represent a better way to develop your apps.

In many cases, people judge the use of a monolith wrongly. I always hear developers complaining that their monolithic apps are challenging to maintain. The truth is that the problem is probably not the app being a monolith. Messy code is likely the main cause of making the app difficult to maintain. Or the fact that developers mixed responsibilities and didn't use abstractions properly could be why the app become difficult to maintain. But a monolithic app doesn't necessarily need to be messy. With software evolution, there are situations in which a monolithic approach no longer works, so we need to find alternatives.

A.2 Using a service-oriented architecture

In this section, we discuss service-oriented architecture. We'll use the example from section 1.1 to prove that a monolithic approach has limitations and that, in some circumstances, we need to use a different style to design your app. You'll learn how a service-oriented architecture solves the presented issues, but we'll also discuss the difficulties this new approach adds to the app's development.

Let's go back to our case with the app for selling books. We have four main functionalities covered by the app: products, deliveries, billing, and customers. What often happens in real-world apps is that not all features consume the resources equally. Some consume more than others, possibly because they're more complex or used more often.

We cannot decide that only a part of the app should be scaled with a monolithic app. In our case, we either scale all four features or none of them. To manage the resources better, we'd like to scale only the features that really need more resources and avoid scaling the others (figure A.4).

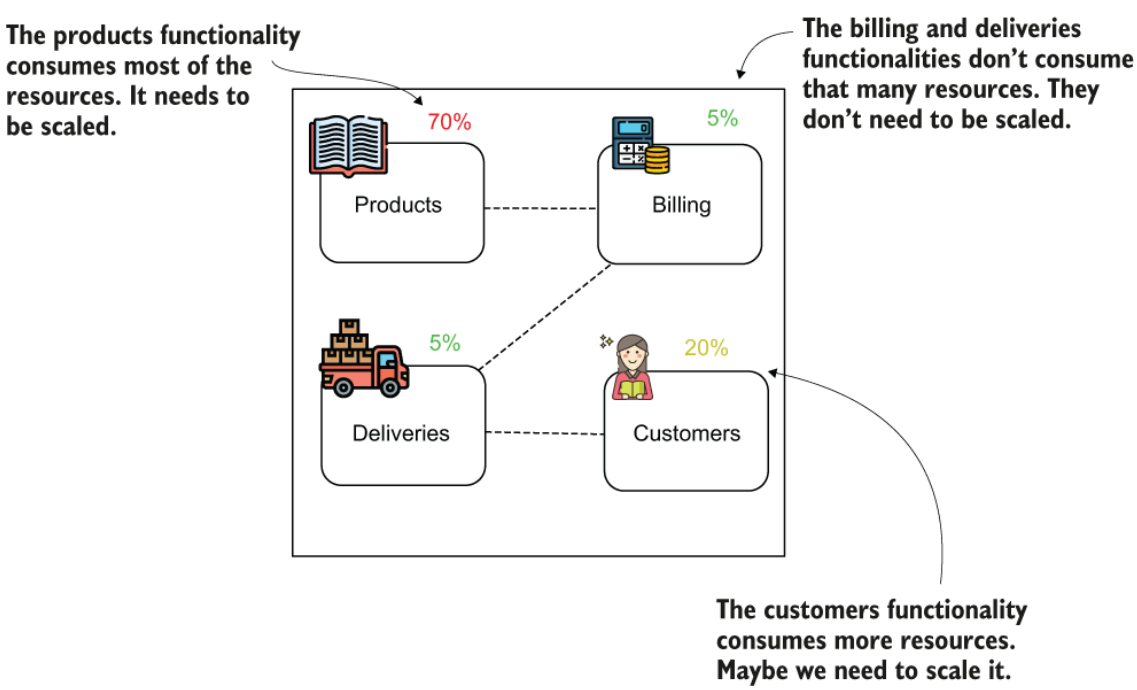

Figure A.4 Some features are more intensively used than others. For this reason, these features consume more resources and need to be scaled.

Can we do something to enable us only to scale the products feature but not the others? Yes, we can split the monolith into multiple services. We'll change the app's architecture from a monolith to a service-oriented architecture (SOA). Instead of having just one process for all the features in a SOA, we have multiple processes implementing the features. We can then decide to scale only the service implementing the feature that needs more resources (figure A.5).

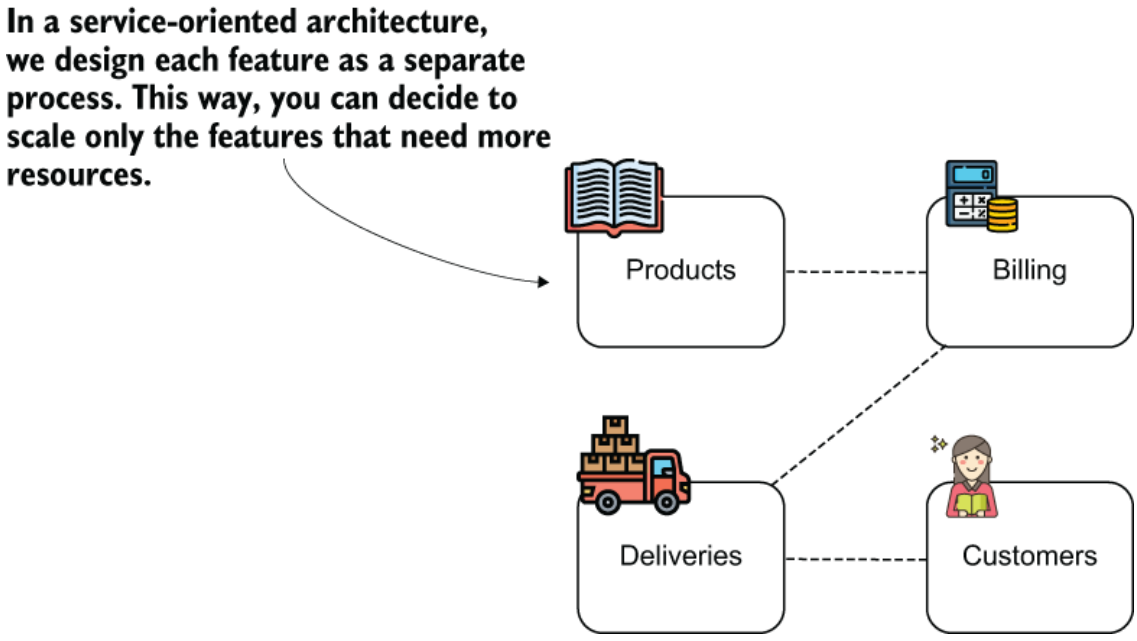

Figure A.5 In a SOA, each feature is an independent process. This way, you can decide to scale just the features that need more resources.

A SOA also has the advantage of having the responsibilities better isolated: now you know you have a dedicated app for the billing and a dedicated app for the deliveries, and so on, and it's easier to keep the implementations decoupled and more cohesive. This is beneficial for the maintainability of the system. As a consequence, it's also easier to manage the teams working on the system because you can offer different teams specific services on which they work, instead of having multiple teams working on the same app (figure A.6).

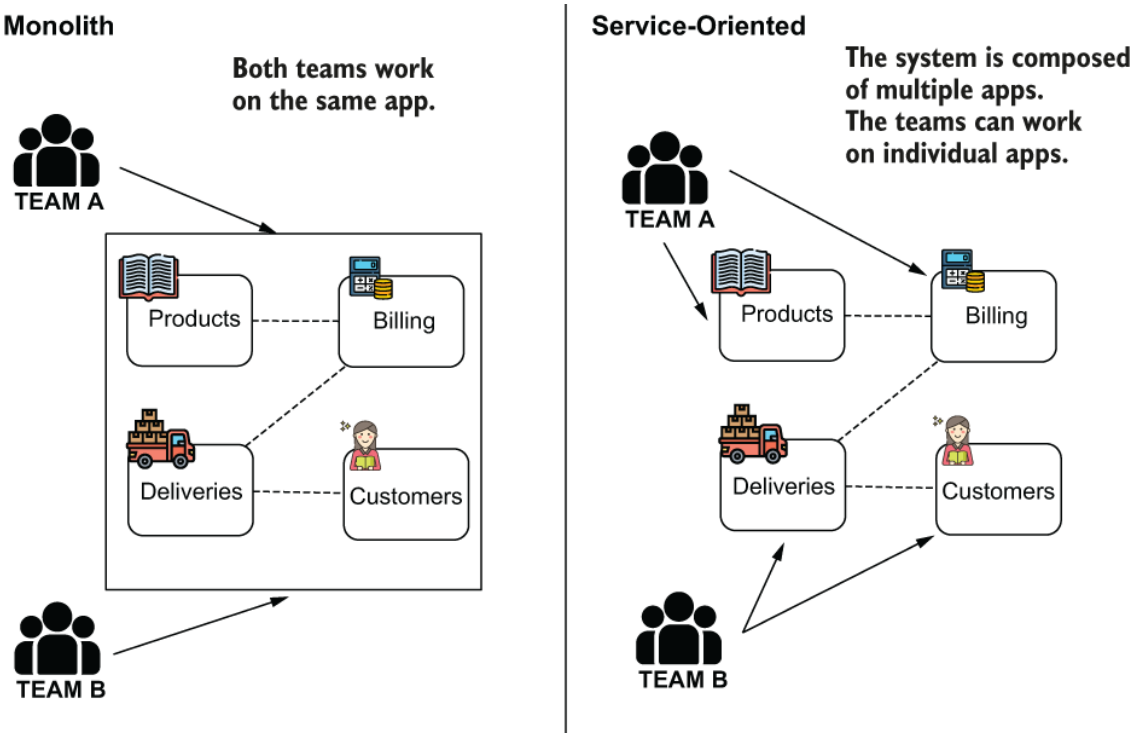

Figure A.6 A monolithic system consists of only one app, so if you have multiple teams working on the system, they all work on the same app. This approach requires more coordination. In a SOA where the system is composed of multiple apps, each team can work on different apps. This way, they need to coordinate less.

At first glance, it might look easy. With all these advantages, why didn't we have all the apps like this from the beginning? Why even bother saying a monolith is still a solution in some cases? To understand the answers to these questions, let's discuss the complexities you introduce using a SOA. Here are some domains in which we encounter different issues when using SOAs:

1. Communication among the services
2. Security
3. Data persistence

4. Deployment

Let's look at some examples.

A.2.1 Complexity caused by communication among services

The functionalities still need to communicate to implement the business logic flow. Earlier, with a monolithic approach, they were part of the same app, which made linking two features easy with a method call. But now that we have different processes, this is more complex.

Features now need to communicate via the network. One of the essential principles you need to remember is that the network isn't entirely reliable. Many fall into the trap of forgetting to consider what happens if, at some point, the communication between two components breaks. Unfortunately, unlike a monolithic approach, any call between two components can fail at some point in a SOA. Depending on the app, developers use different techniques or patterns to solve this issue, like repeating calls, circuit breakers, or caches.

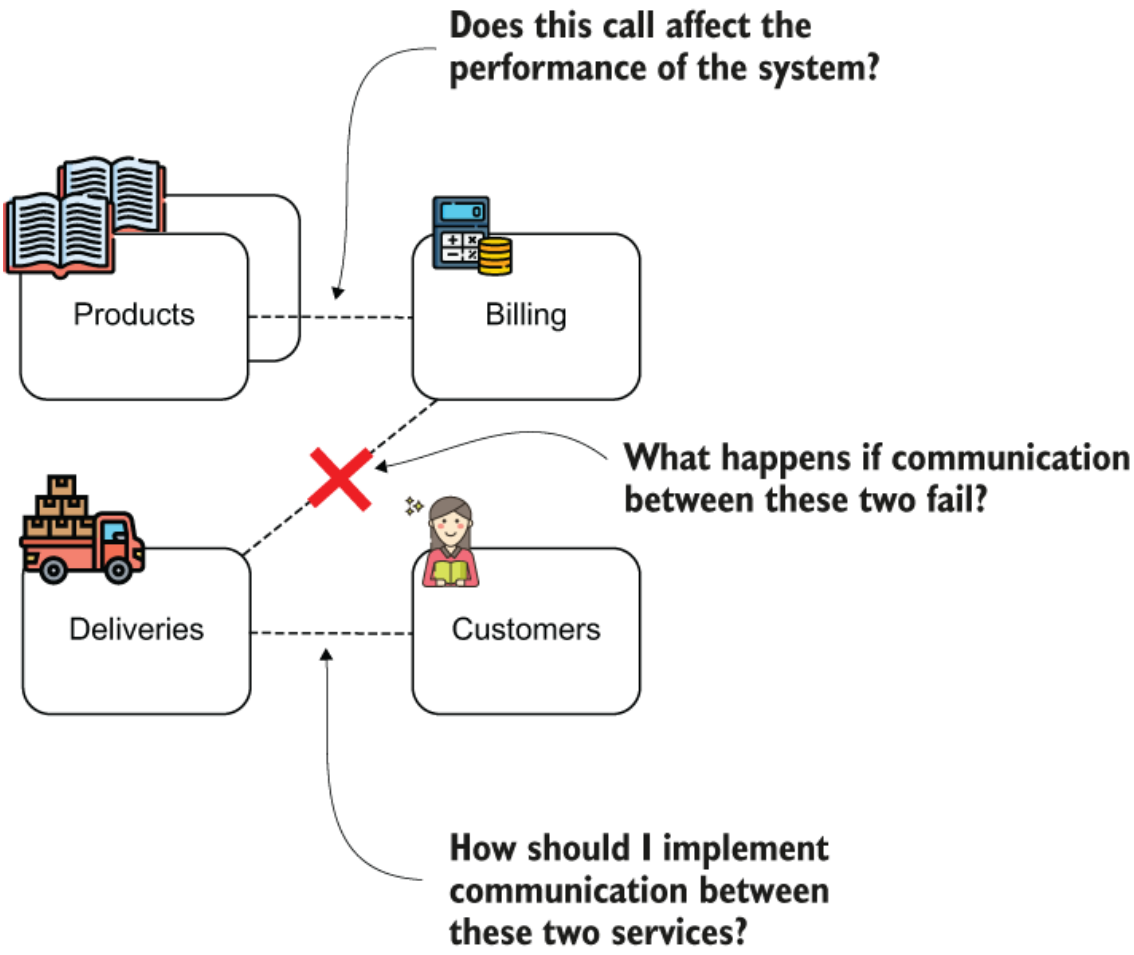

Figure A.7 Communication between services adds complexity to the system. We need to decide how to implement the communication between

two services. We also need to understand what could happen if the communication fails and how to solve potential problems caused by malfunctioning communication.

A second aspect to consider is that there are many options to establish communication among the services (figure A.7). You could use REST services, GraphQL, SOAP, gRPC, JMS message brokers, Kafka, and so on. Which is the best approach? Of course, in any situation one or more of these approaches is fine. You'll find long debates and discussions in many books on how to choose the right fit for typical scenarios.

A.2.2 Complexity added to the security of the system

By splitting our functionalities into separate services, we also introduce complexity in security configurations. These services exchange messages via the network, which might expose information. Sometimes we want pieces of the data exchanged not to be seen at all (like passwords, banking card details, or other personal data). We now need to encrypt these details before sending them. Even if we don't care if someone can see the exchanged details, in most cases we don't want anyone to be able to change them while they flow from one component to another (figure A.8).

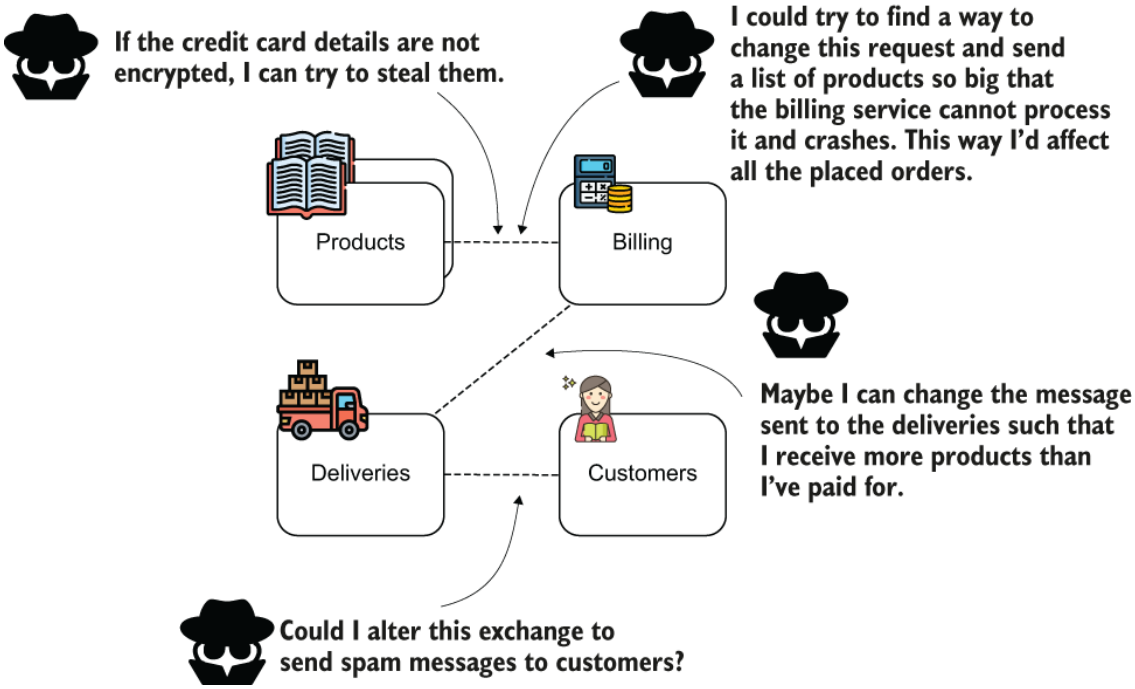

Figure A.8 In a SOA, features are separate services and communicate over the network. This aspect introduces many vulnerable points that developers need to consider when building the app.

A.2.3 Complexity added for the data persistence

In most cases, an app needs a way to store data. Databases are a popular way persistence is implemented in apps. With a monolithic approach, an app had one database to store the data, as presented in figure A.9. We called this a *three-tier architecture* because it consists of three tiers: the client, the backend, and the database used for persistence.

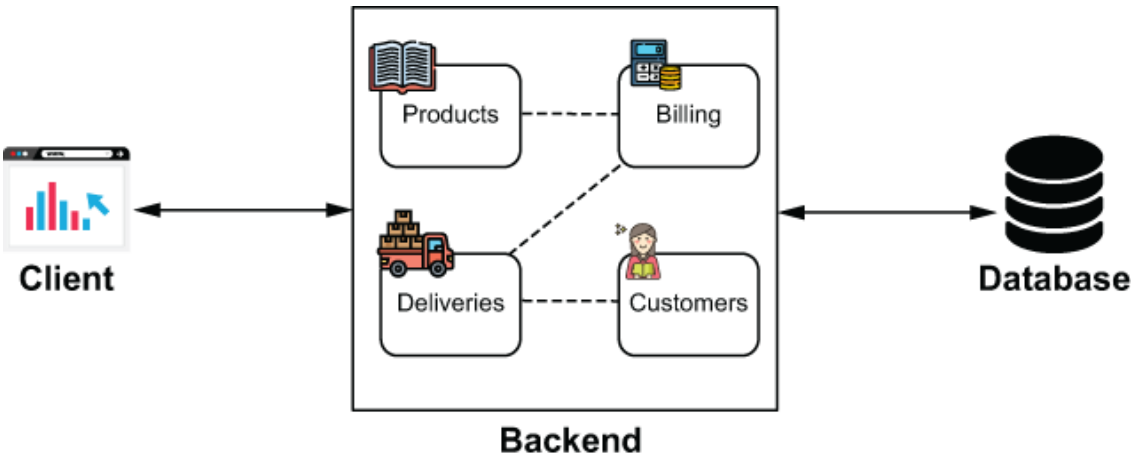

Figure A.9 With a monolithic approach, you only have one application and usually one database. The system is simple and can be easily visualized and understood.

With SOA, you now have multiple services that need to store data. And with more services, you also have more design options. Should you use just one database shared by all the services? Should you have one database for each service? Figure A.10 visualizes these options.

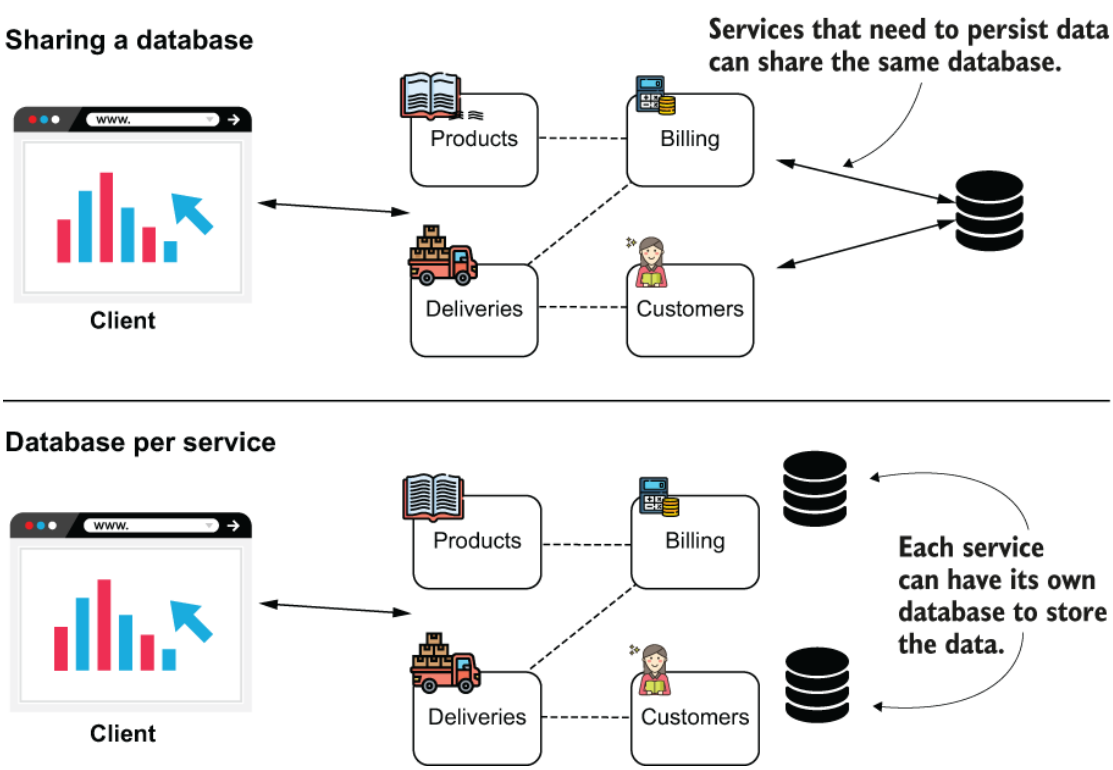

Figure A.10 In a SOA, you can decide that more services share the same database or have an individual database per service. Having various alternatives, each with its benefits and drawbacks, makes the persistence layer's design in SOA more difficult.

Most believe that sharing a database is bad practice. From my own experience with splitting a monolith into multiple services, I can tell you that having a shared database can become a deployment nightmare. But having individual databases per service also implies difficulties. As you'll see when we discuss transactions, it's much easier to assure data consistency with one database. When having more independent databases, it's challenging to make sure the data remains consistent among all.

A.2.4 Complexity added in the deployment of the system

Maybe the easiest challenge to observe is that we add a lot of complexity to the system's deployment. We also have more services now, but as you learned from the previous paragraphs, you might have multiple databases as well. When you also consider that securing the system will add even more configurations, you see how much more complex the deployment of the system becomes.

Why does a monolith have a negative connotation?

You can see that the SOAs aren't necessarily easy, so you might wonder why monolithic architecture tends to be associated with something negative. The reality is that for some systems, a monolith makes more sense than SOA.

My opinion is that the negative connotation of monolithic architecture comes from the fact that it represents old systems. In most cases, old systems were implemented before anyone was concerned about clean coding and design principles. We now consider all these principles to make sure we write maintainable code.

It might feel strange to look back to the times when they didn't exist, and sometimes I've even seen developers blame those who started the implementation of such old systems when problems arise. But the truth is that it's not the fault of the folks who used the tools and practices that everyone considered the best at that time.

Today, many developers associate messy and poorly written code with a monolith concept. However, monolithic apps can be modular, and their code can be clean, while service-oriented apps can be messy and poorly designed.

1.3 From microservices to serverless

In this section, we'll discuss microservices. In this book, I refer to microservices here and there, and I'd like you to be at least aware of what they mean. Microservices are a particular implementation of the SOA. A microservice usually is designed with one responsibility and has its own persistence capability (it doesn't share databases).

With time, the way we deploy apps changed. The software architecture is not only related to the functionality of the app. A wise software architect knows to adapt the system's architecture to both the way teams work on the system and how the system is deployed. You might have heard about what we call the DevOps movement, which implies both how we deploy

software as well as how we work on software development. Today, we deploy the apps in the cloud using virtual machines or containerized environments, and these approaches generally implied the need for making apps smaller. Of course, evolution came with another incertitude: how small should a service be? Many debated this question in books, articles, and discussions.

The minimization of services went so far that today we can implement a short functionality with only a few lines of code and deploy it in an environment. An event like an HTTP request, a timer, or a message triggers this functionality and makes it execute. We call these small implementations *serverless* functions. The term "serverless" doesn't imply that the function doesn't execute on a server. But because everything regarding the development is hidden and we only case for the code that implements its logic and the events that trigger it, it merely looks like no server exists.

A.4 Further reading

Software architecture and its evolution is such a fantastic and complex subject. I don't think there'll ever be too many books to cover this subject thoroughly. I've added this discussion to the book to help you understand the references I'll make to these notions. Still, you might want to go deeper into the subjects, so here's a list of books from my shelf. The books are in the order I recommend you read them.

1. *Microservices in Action*, by Morgan Bruce and Paulo A. Pereira (Manning, 2018), is an excellent book you can start with when learning microservices. In the book, you'll find all the microservices fundamentals subjects discussed with useful examples.
2. *Microservices Patterns*, by Chris Richardson (Manning, 2018), is a book I recommend you continue with after thoroughly reading *Microservices in Action*. The author presents a pragmatic approach on how to develop production-ready apps using microservices.
3. *Spring Microservices in Action*, by John Carnell and Illary Huaylupo Sánchez (Manning, 2020), helps you better understand how to apply Spring to build microservices.

4. *Microservices Security in Action*, by Prabath Siriwardena and Nuwan Dias (Manning 2020), goes into detail with what applying security with a microservice architecture means. Security is a crucial aspect of any system, and you always need to consider it from the development process's early stages. The book explains security from the ground up, and reading it will give you a better understanding the aspects you need to be aware of in regards to security for microservices.
5. *Monolith to Microservices*, by Sam Newman (O'Reilly Media, 2020), treats patterns for transforming a monolithic architecture into microservices. The book also discusses whether you need to use microservices and how to decide this.