Microservices in .NET Core

SECOND EDITION

Christian Horsdal Gammelgaard

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Microservices in .NET Core
Second Edition
Version 4

Copyright 2020 Manning Publications

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

©Manning Publications Co. To comment go to liveBook

welcome

Thank you for buying the MEAP of the second edition of *Microservices in .NET Core*.

This is a book for developers and architects interested in getting better at building and designing microservices. It is a practical book that will show the details of how to develop robust, scalable, and flexible microservices. It's a book that takes a developer's point of view. It shows you the code, but it also thoroughly teaches you the ideas behind the code.

For this second edition, I use different tooling than the first time around. Like in the first edition, I'm still using .NET Core. I updated the framework and am taking advantage of the benefits of containerization this time around. The first edition of this book used the Nancy web framework, but, sadly, I feel Nancy has lost its momentum, but Microsoft's MVC framework has gotten better and better. So, for this edition, I am using Microsoft's MVC framework. Furthermore, I decided it was time to containerize the microservices. In this edition, all microservices are wrapped up in Docker containers and deployed to Kubernetes.

The book starts with an introduction to microservices—what they are, why they are helpful, and when they look like. The second part digs into how you design microservices and their collaboration, how to deal with data, and how to write good automated tests for microservices. The third part builds upon the previous parts and shows you how to build re-usable code for dealing with things that cut across all your microservices. This will help you get new microservice off the ground quickly, which you will be doing often once you really get going with microservices. Finally, in the fourth part, we will explore how to build end-user applications on top of a system of many, many microservices.

Feedback is essential for creating a high-quality book. Your feedback is essential for me. Since this is an early access edition, I will have time to incorporate your feedback, and I encourage you to give it! Please use the liveBook discussion forum for this book to post your feedback.

Thanks, Christian

©Manning Publications Co. To comment go to liveBook

brief contents

©Manning Publications Co. To comment go to liveBook

1 Microservices at a glance

This chapter covers

- Understanding microservices and their core characteristics
- Examining the benefits and drawbacks of microservices
- An example of microservices working in concert to serve a user request
- Using ASP.NET Core for a simple application

In this chapter, I'll explain what microservices are and demonstrate why they're interesting. We'll also look at the six characteristics of a microservice. Finally, I'll introduce you to the most important technologies we'll use in this book: .NET Core and MVC Core.

1.1 What is a microservice?

A *microservice* is a service with one, and only one, very narrowly focused capability that a remote API exposes to the rest of the system. For example, think of a system for managing a warehouse. If you broke down its capabilities, you might come up with the following list:

- Receive stock arriving at the warehouse.
- Determine where new stock should be stored.
- Calculate placement routes inside the warehouse for putting stock into the right storage units.
- Assign placement routes to warehouse employees.
- Receive orders.
- Calculate pick routes in the warehouse for a set of orders.
- Assign pick routes to warehouse employees.

Let's consider how the first of these capabilities—receive stock arriving at the warehouse—would be implemented as a microservice. We'll call it the *Receive Stock microservice*:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

1. A request to receive and log new stock arrives over HTTP. This might come from another microservice or perhaps from a web page that a foreman uses to register stock arrivals. The responsibility of Receive Stock microservice is to handle such requests by validating the request and correctly registering the new stock in a data store.
2. A response is sent back from the Receive Stock microservice to acknowledge that the stock has been received.

Figure 1.1 shows the Receive Stock microservice receiving a request from another collaborating microservice.

Figure 1.1 The Receive Stock microservice exposes an API to be used when new stock arrives. Other microservices can call that API as indicated by the arrow. The receive stock microservice is responsible for registering all received stock in a data store.

Each little capability in the system is implemented as an individual microservice. Every microservice in a system

- Runs in its own separate process
- Can be deployed on its own, independently of the other microservices
- Has its own dedicated data store
- Collaborates with other microservices to complete its own action

It's also important to note that a microservice doesn't need to be written in the same programming language (C#, Java, Erlang, and so on) as one it collaborates with. They just need to know how to communicate with each other. Some may communicate via a service bus or a binary protocol like gRPC, depending on system requirements; but often microservices do communicate over HTTP.

NOTE	This book focuses on implementing microservices in .NET using C# and ASP.NET Core. The microservices I'll show you are small, tightly focused ASP.NET Core applications that collaborate over HTTP.

1.2 What is a microservices architecture?

This book focuses on designing and implementing individual microservices, but it's worth noting that the term *microservices* can also be used to describe an *architectural style* for an entire system consisting of many microservices. Microservices *as an architectural style* is a lightweight form of service-oriented architecture (SOA) where the services are tightly focused on doing one thing each and doing it well. A system with a microservices architecture is a distributed system with a (probably large) number of collaborating microservices.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

The microservices architectural style is quickly gaining in popularity for building and maintaining complex server-side software systems. And understandably so: microservices offer a number of potential benefits over both more traditional service-oriented approaches and monolithic architectures. Microservices, when done well, are malleable, scalable, and robust, and they allow for systems that do well on all four of the key metrics identified by Nicole Forsgren et. al. in Accelerate[1] and the DORA state of DevOps reports[2], namely:

1. Deployment frequency
2. Lead time for changes
3. Time to restore service
4. Change failure rate

This combination often proves elusive for complex software systems. Furthermore it is, as documented by Forsgren et. al., a reliable predictor of software delivery performance.

In this book you will learn how to design and implement malleable, scalable and robust microservices which form systems that deliver on all four of the key metrics above.

1.2.1 Microservice characteristics

I've said that a microservice is *a service with a very narrowly focused capability*, but what exactly does that mean? Well, there's not a broadly accepted definition in the industry of precisely what a microservice is.[3] We can, however, look at what generally characterizes a microservice. I've found there to be six core microservice characteristics:

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A microservice consists of one or more processes.
- A microservice owns its own data store.
- A small team can maintain a few handfuls of microservices.
- A microservice is replaceable.

This list of characteristics should help you recognize a well-formed microservice when you see one, and it will also help you scope and implement your own microservices. By incorporating these characteristics, you'll be on your way to getting the best from your microservices and producing *a malleable, scalable, and robust system* as a result. Throughout this book, I'll show how these characteristics should drive the design of your microservices and how to write the code that a microservice needs to fulfill them. Now, let's look briefly at each characteristic in turn.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

RESPONSIBLE FOR A SINGLE CAPABILITY

A microservice is *responsible for one and only one capability* in the overall system. We can break this statement into two parts:

- A microservice has a single responsibility.
- That responsibility is for a capability.

The Single Responsibility Principle has been stated in several ways. One traditional form is, "A class should have only one reason to change."[4] Although this way of putting it specifically mentions a *class*, the principle turns out to apply beyond the context of a class in an object-oriented language. With microservices, we apply the Single Responsibility Principle at the service level.

Another, newer, way of stating the single responsibility principle, also from Robert C. Martin, is as follows: "Gather together the things that change for the same reasons. Separate those things that change for different reasons."[5] This way of stating the principle applies to microservices: a microservice should implement exactly one capability. That way, the microservice will have to change only when there's a change to that capability. Furthermore, you should strive to have the microservice fully implement the capability, so that only one microservice has to change when the capability is changed.

There are two types of capabilities in a microservice system:

- A *business capability* is something the system does that contributes to the purpose of the system, like keeping track of users' shopping carts or calculating prices. A good way to tease apart a system's separate business capabilities is to use domain-driven design.
- A *technical capability* is one that several other microservices need to use—integration to some third-party system, for instance. Technical capabilities aren't the main drivers for breaking down a system to microservices; they're only identified when you find several business-capability microservices that need the same technical capability.

NOTE Defining the scope and responsibility of a microservice will be covered in chapter 4.

INDIVIDUALLY DEPLOYABLE

A microservice should be *individually deployable*. When you change a particular microservice, you should be able to deploy that changed microservice to the production environment without deploying (or touching) any other part of your system. The other microservices in the system should continue running and working during the deployment of the changed microservice and continue running once the new version is deployed.

Consider an e-commerce site. Whenever a change is made to the Shopping Cart microservice,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

you should be able to deploy just that microservice, as illustrated in figure 1.2. Meanwhile, the Price Calculation microservice, the Recommendation microservice, the Product Catalog microservice, and others should continue working and serving user requests.

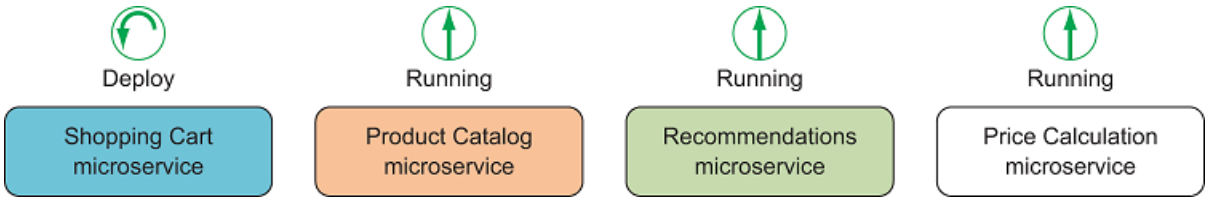

Figure 1.2 Other microservices continue to run while the Shopping Cart microservice is being deployed.

Being able to deploy each microservice individually is important because in a microservice system, there are many microservices, and each one may collaborate with several others. At the same time, development work is done on some or all of the microservices in parallel. If you had to deploy all or groups of them in lockstep, managing the deployments would quickly become unwieldy, typically resulting in infrequent and big, risky deployments. This is something you should definitely avoid. Instead, you want to be able to deploy small changes to each microservice frequently, resulting in small, low-risk deployments.

To be able to deploy a single microservice while the rest of the system continues to function, the build process must be set up with the following in mind:

- Each microservice must be built into separate artifacts.
- The deployment process must also be set up to support deploying microservices individually while other microservices continue running. For instance, you might use a rolling deployment process where the microservice is deployed to one server at a time, in order to reduce downtime.

The fact that you want to deploy microservices individually affects the way they interact. Changes to a microservice's interface usually must be backward compatible so other existing microservices can continue to collaborate with the new version the same way they did with the old. Furthermore, the way microservices interact must be robust in the sense that each microservice must expect other services to fail once in a while and must continue working as best it can. One microservice failing—for instance, due to downtime during deployment—must not result in other microservices failing, only in reduced functionality or slightly longer processing time.

NOTE Microservice collaboration and robustness will be covered in chapters 4, 5, and 7.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

CONSISTS OF ONE OR MORE PROCESSES

A microservice must run in a separate process, or in separate processes, if it's to remain as independent as possible of other microservices in the same system. The same is true if a microservice is to remain individually deployable. Breaking that down, we have two points:

- Each microservice must run in separate processes from other microservices.
- Each microservice can have more than one process.

Consider a Shopping Cart microservice again. If it ran in the same process as a Product Catalog microservice, as shown in figure 1.3, the Shopping Cart code might cause a side effect in the Product Catalog. That would mean a tight, undesirable coupling between the Shopping Cart microservice and the Product Catalog microservice; one might cause downtime or bugs in the other.

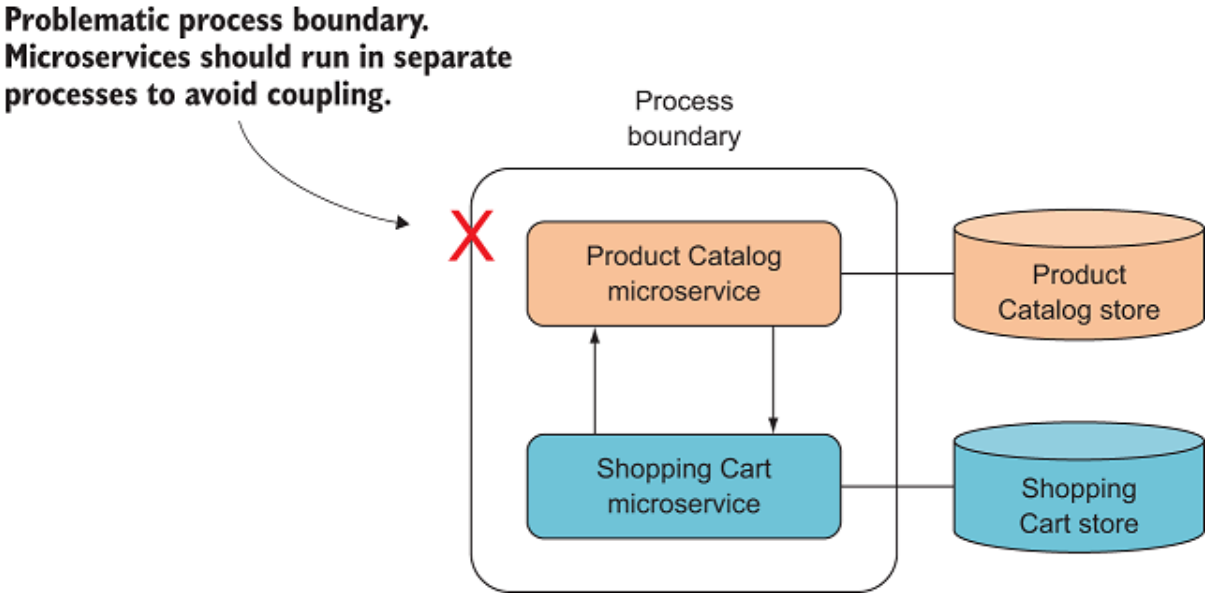

Figure 1.3 Running more than one microservice within a process leads to high coupling between the two: They cannot be deployed individually and one might cause downtime in the other.

Now consider deploying a new version of the Shopping Cart microservice. You'd either have to redeploy the Product Catalog microservice too or need some sort of dynamic code-loading capable of switching out the Shopping Cart code in the running process. The first option goes directly against microservices being individually deployable. The second option is complex and at a minimum puts the Product Catalog microservice at risk of going down due to a deployment to the Shopping Cart microservice.

Speaking of complexity, why should a microservice consist of more than one process? You are, after all, trying to make each microservice as simple as possible to handle.

Let's consider a Recommendation microservice. It implements and runs the algorithms that drive

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

recommendations for your e-commerce site. It also has a database that stores the data needed to provide recommendations. The algorithms run in one process, and the database runs in another. Often, a microservice needs two or more processes so it can implement everything (such as data storage and background processing) it needs in order to provide a capability to the system.

OWNS ITS OWN DATA STORE

A microservice owns the data store where it stores the data it needs. This is another consequence of a microservice's scope being a complete capability. Most business capabilities require some data storage. For instance, a Product Catalog microservice needs some information about each product to be stored. To keep Product Catalog loosely coupled with other microservices, the data store containing the product information is completely owned by the microservice. The Product Catalog microservice decides how and when the product information is stored. As illustrated in 1.4, other microservices, such as Shopping Cart, can only access product information through the interface to Product Catalog and never directly from the Product Catalog data store.

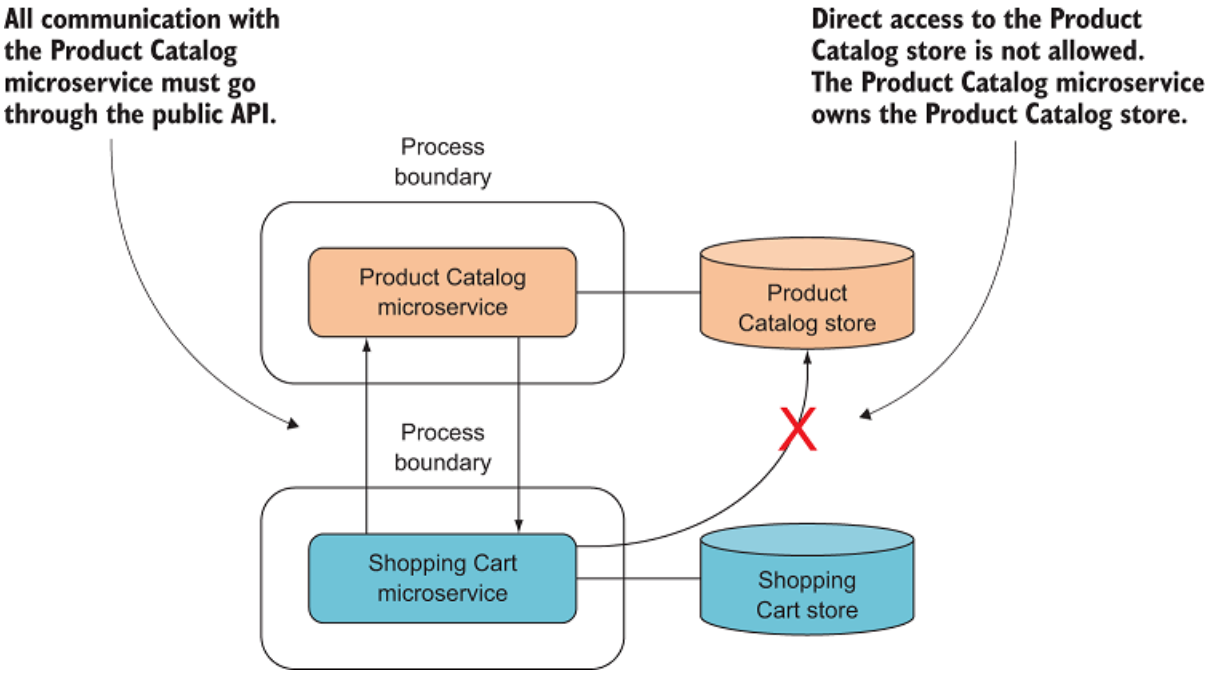

Figure 1.4 One microservice can't access another's data store.

The fact that each microservice owns its own data store makes it possible to use different database technologies for different microservices depending on the needs of each microservice. The Product Catalog microservice, for example, might use SQL Server to store product information; the Shopping Cart microservice might store each user's shopping cart in Redis; and the Recommendations microservice might use an ElasticSearch index to provide recommendations. The database technology chosen for a microservice is part of the implementation and is hidden from the view of other microservices.

This approach allows each microservice to use whichever database is best suited for the job,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

which can also lead to benefits in terms of development time, performance, and scalability. The obvious downside is the need to administer, maintain, and work with more than one database, if that's how you choose to architect your system. Databases tend to be complicated pieces of technology, and learning to use and run one reliably in production isn't free. When choosing a database for a microservice, you need to consider this trade-off. But one benefit of a microservice owning its own data store is that you can swap out one database for another later.

NOTE Data ownership, access, and storage will be covered in chapter 5.

MAINTAINED BY A SMALL TEAM

So far, I haven't talked much about the size of a microservice, even though the *micro* part of the term indicates that microservices are small. I don't think it makes sense to discuss the number of lines of code that a microservice should have, or the number of requirements, use cases, or function points it should implement. All that depends on the complexity of the capability provided by the microservice.

What does make sense, though, is considering the amount of work involved in maintaining a microservice. The following rule of thumb can guide you regarding the size of microservices: *a small team of people—five, perhaps—should be able to maintain a few handfuls of microservices* . Here, *maintaining a microservice* means dealing with all aspects of keeping it healthy and fit for purpose: developing new functionality, factoring out new microservices from ones that have grown too big, running it in production, monitoring it, testing it, fixing bugs, and everything else required.

REPLACEABLE

For a microservice to be *replaceable*, it must be able to be rewritten from scratch within a reasonable time frame. In other words, *the team maintaining the microservice should be able to replace the current implementation with a completely new implementation and do so within the normal pace of their work*. This characteristic is another constraint on the size of a microservice: if a microservice grows too large, it will be expensive to replace; but if it's kept small, rewriting it is realistic.

Why would a team decide to rewrite a microservice? Perhaps the code is a big jumble and no longer easily maintainable. Perhaps it doesn't perform well enough in production. Neither is a desirable situation, but changes in requirements over time can result in a codebase that it makes sense to replace rather than maintain. If the microservice is small enough to be rewritten within a reasonable time frame, it's OK to end up with one of these situations from time to time. The team does the rewrite based on all the knowledge obtained from writing the existing implementation and keeping any new requirements in mind.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Now that you know the characteristics of microservices, let's look at their benefits, costs, and other considerations.

1.3 Why microservices?

Building a system from microservices that adhere to the characteristics outlined in the previous section has some appealing benefits: they're malleable, scalable, and robust, and they allow a short lead time from start of implementation to deployment to production. This adds up to doing well on all of the four key metrics outlined in Accelerate[6]. These benefits are realized because, when done well, microservices

- Enable continuous delivery
- Allow for an efficient developer workflow because they're highly maintainable
- Are robust by design
- Can scale up or down independently of each other

Let's talk more about these points.

1.3.1 Enabling continuous delivery

The microservices architectural style takes continuous delivery into account. It does so by focusing on services that

- Can be developed and modified quickly
- Can be comprehensively tested by automated tests
- Can be deployed independently
- Can be operated efficiently

These properties enable continuous delivery, but this doesn't mean continuous delivery follows from adopting a microservices architecture. The relationship is more complex: practicing continuous delivery becomes easier with microservices than it typically is with more traditional SOA. On the other hand, fully adopting microservices is possible only if you're able to deploy services efficiently and reliably. Continuous delivery and microservices complement each other.

The benefits of continuous delivery are well known. They include increased agility on the business level, reliable releases, risk reduction, and improved product quality.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **What is continuous delivery?**

Continuous delivery is a development practice where the team ensures that the software can always be deployed to production quickly at any time. Releasing to market remains a business decision, but teams that practice continuous delivery string to deploy to production often and to deploy newly developed software shortly after it hits source control.

There are two main requirements for continuous delivery. First, the software must always be in a fully functional state. To achieve that, the team needs a keen focus on quality. This leads to a high degree of test automation and to developing in very small increments. Second, the deployment process must be repeatable, reliable, and fast in order to enable frequent production deployments. This part is achieved through full automation of the deployment process and a high degree of insight into the health of the production environment.

Although continuous delivery takes a good deal of technical skill, it's much more a question of process and culture. This level of quality, automation, and insight requires a culture of close collaboration among all parties involved in developing and operating the software, including businesspeople, developers, information security experts, and system administrators. In other words, it requires a DevOps culture where development, operations, and other groups collaborate and learn from each other.

Continuous delivery goes hand in hand with microservices. Without the ability to deploy individual microservices quickly and cheaply, implementing a system of microservices will quickly become expensive. If microservice deployment isn't automated, the amount of manual work involved in deploying a full system of microservices will be overwhelming.

Along with continuous delivery comes a DevOps culture, which is also a prerequisite for microservices. To succeed with microservices, everybody must be invested in making the services run smoothly in production and in creating a high level of transparency into the health of the production system. This requires the collaboration of people with operations skills, people with development skills, people with security skills, and people with insight into the business domain, among others.

This book won't focus on continuous delivery or DevOps, but it will take for granted that the environment in which you develop microservices uses continuous delivery. The services built in this book can be deployed to any cloud or to on-premise servers using any number of deployment-automation technologies capable of handling .NET. This book does cover the implications of continuous delivery and DevOps for individual microservices. In part 3, we'll go into detail about how to build a platform that handles a number of the operational concerns that all microservices must address.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

As an example of deployment we will create a Kubernetes environment in Microsofts Azure cloud in chapter 3 and we will deploy a microservice to it. Throughout the chapters following chapter 3 we will continue to use that Kubernetes environment as an example and continue to deploy microservices to it.

1.3.2 High level of maintainability

Well-factored and well-implemented microservices are highly maintainable from a couple of perspectives. *From a developer perspective*, several factors play a part in making microservices maintainable:

- Each well-factored microservice provides *a single capability*. Not two—just one.
- A microservice owns its own data store. No other services can interfere with a microservice's data store. This, combined with the typical size of the codebase for a microservice, means you can understand a complete service all at once.
- Well-written microservices can (and should) be comprehensibly covered by automated tests.

From an operations perspective, a couple of factors play a role in the maintainability of microservices:

- A small team can maintain a few handfuls of microservices. Microservices must be built to be operated efficiently, which implies that you should be able to easily determine the current health of any microservice.
- Each microservice is individually deployable.

It should follow that issues in production can be discovered in a timely manner and be addressed quickly, such as by scaling out the microservice in question or deploying a new version of the microservice. The characteristic that a microservice owns its own data store also adds to its operational maintainability, because the scope of maintenance on the data store is limited to the owning microservice.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Favor lightweight**

Because every microservice handles a single capability, microservices are by nature fairly small both in their scope and in the size of their codebase. The simplicity that follows from this limited scope is a major benefit of microservices.

When developing microservices, it's important to avoid complicating their codebase by using large, complicated frameworks, libraries, or products because you think you may need their functionality in the future. Chances are, this won't be the case, so you should prefer smaller, lightweight technologies that do what the microservice needs right now. Remember, a microservice is replaceable; you can completely rewrite a microservice within a reasonable budget if at some point the technologies you used originally no longer meet your needs.

1.3.3 Robust and scalable

A microservices-based distributed architecture allows you to scale out each microservice individually based on where bottlenecks occur. Furthermore, microservices favor asynchronous event-based collaboration and stress the importance of fault tolerance wherever synchronous communication is needed. When implemented well, these properties result in highly available, highly scalable systems.

1.4 Costs and downsides of microservices

Significant costs are associated with choosing a microservices architecture, and these costs shouldn't be ignored:

- Microservice systems are distributed systems. The costs associated with distributed systems are well known: they can be harder to reason about and harder to test than monolithic systems, and communication across process boundaries or across networks is orders of magnitude slower than in-process method calls.
- Microservice systems are made up of many microservices, each of which has to be developed, deployed, and managed in production. This means you'll have many deployments and a complex production setup.
- Each microservice is a separate codebase. Consequently, refactorings that move code from one microservice to another are painful. You need to invest in getting the scope of each microservice just right.

Before jumping head first into building a system of microservices, you should consider whether the system you're implementing is sufficiently complex to justify the associated overhead.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Do microservices perform?**

One question that always seems to pop up in discussions of whether to use microservices is whether a system built with microservices will be as performant as a system that's not. The argument against is that if the system is built from many, many collaborating microservices, every user request will involve several microservices, and the collaboration between these microservices will involve remote calls between them. What happens when a user request comes in? Do you chain together a long series of remote calls going from one microservice to the next? Considering that remote calls are orders of magnitude slower than calls inside a process, this sounds slow.

The problem with this argument is the idea that you'd be making roughly the same calls between different parts of the system as you would if everything were in one process. First, as we'll learn in chapters 5 and 7, the interaction between microservices should be much less fine-grained than calls within a process tend to be. Second, as we'll discuss in chapters 5 and 6, you'll prefer event-based asynchronous collaboration over making synchronous remote calls, and you'll store copies of the same data in several microservices to make sure it's available where it's needed. All in all, these techniques drastically reduce the need to make remote calls while a user is waiting. Moreover, the fine-grained nature of microservices enables you to scale out the specific parts of the system that get congested.

There isn't a simple yes or no answer as to whether microservices perform well. What I can say is that a well-designed microservice system can easily meet the performance requirements of many, if not most, systems.

1.5 Greenfield vs. brownfield

Should you introduce microservices from the get-go on a new project, or are they only relevant for large, existing systems? This question tends to come up in discussions about microservices.

The microservices architectural style has grown out of the fact that many organizations' systems started out small but have grown big over time. Many of these systems consist of a single large application—a monolith that often exposes the well-known disadvantages of big, monolithic systems:

- Coupling is high throughout the codebase.
- There's hidden coupling between subcomponents: coupling that is now obvious at first glance can string from knowledge implicit in the code about how certain strings are formatted, how certain columns in a databases are used, and so on.
- Deploying the application is a lengthy process that may involve several people and

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

system downtime.
- The system has a one-size-fits-all architecture intended to handle the most complex components. If you insist on architectural consistency across the monolith, the least complex parts of the system will be over-engineered. This is true of layering, technology choices, chosen patterns, and so on.

The microservices architectural style arose as a result of solving these problems in existing monolithic systems. If you repeatedly split subcomponents of a monolith into ever-smaller and more manageable parts, microservices are eventually created. [7]

On the other hand, new projects are started all the time. Are microservices irrelevant for these greenfield projects? That depends. Here are some questions you need to ask yourself:

- Would this system benefit from the ability to deploy subsystems separately?
- Can you build sufficient deployment automation?
- Are you sufficiently knowledgeable about the domain to properly identify and separate the system's various independent business capabilities?
- Is the system's scope large enough to justify the complexity of a distributed architecture?
- Is the system's scope large enough to justify the cost of building the deployment automation?
- Will the project survive long enough to recover the up-front investment in automation and distribution?

Some greenfield projects meet these criteria and may benefit from adopting a microservices architecture from the outset.

1.6 Code reuse

Adopting a microservices architecture leads to having many services, each of which has a separate codebase that you'll have to maintain. It's tempting to look for code reuse across services in the hope that you can reduce the maintenance effort; but although there's an obvious potential benefit to code reuse, pulling code out of a service and into a reusable library incurs a number costs that may not be immediately apparent:

- The service now has one more dependency that you must understand in order to understand the complete service. This isn't to say that there's more code to comprehend; but by moving code out of the service and into a library, you move the code further away, making simple code navigation slower and refactoring more difficult.
- The code in the new library must be developed and maintained with multiple use cases in mind. This tends to take more effort than developing for just one use case.
- The shared library introduces a form of coupling between the services using it. Updates to the library driven by the needs of service A may not be needed in service B. Should service B update to the new version of the library even though it's not strictly necessary? If you upgrade B, it will have code it doesn't need; and, worse, B will run the risk of errors caused by that code. If you don't upgrade, you'll have several versions of the library in production, further complicating maintenance of the library. Both cases incur some complexity, either in service B or in the combined service landscape.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

These points apply particularly to business code. Business code should almost never be reused across microservices. That type of reuse leads to harmful coupling between microservices.

With these points in mind, you should be wary of code reuse and only judiciously attempt it. There is, however, a case to be made for reusing infrastructure code that implements technical concerns.

To keep a service small and focused on providing one capability well, you'll often prefer to write a new service from scratch rather than add functionality to an existing service. It's important to do this quickly and painlessly, and this is where code reuse across services is relevant. As we'll explore in detail in part 3 of this book, there are a number of technical concerns that all services need to implement in order to fit well into the overall service landscape. You don't need to write this code for every single service; you can reuse it across services to gain consistency in how these technical aspects are handled and to reduce the effort needed to create a new service.

1.7 Serving a user request: an example of how microservices work in concert

To get a feel for how a microservices architecture works, let's look at an example: a user of an e-commerce website adding an item to their shopping cart. From the viewpoint of the client-side code, a request is fired to the back-end system via an API gateway, and an updated shopping cart along with some price information is returned. This is as simple as the interaction shown in figure 1.5. We'll return to the topic of API gateways in chapter 13.

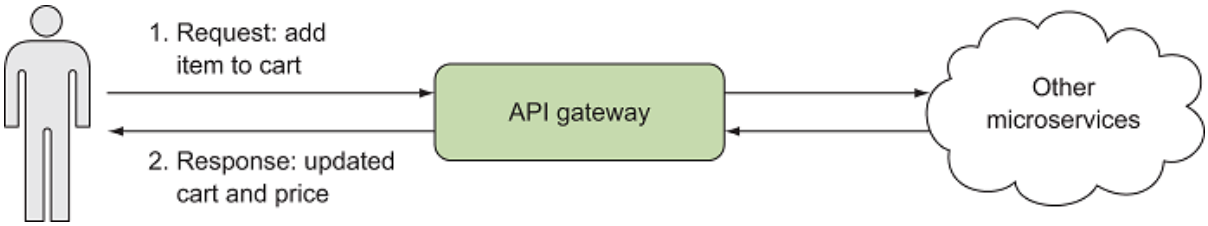

Figure 1.5 When front-end code makes a request to add an item to the shopping cart, it only communicates with the API Gateway microservice. What goes on behind the gateway isn't visible.

This is neither surprising nor exciting. The interesting part is the interactions taking place behind the API Gateway microservice to fulfill the request. To add the new item to the user's shopping cart, the API Gateway uses a few other microservices. Each microservice is a separate process, and in this example they communicate via HTTP requests.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

1.7.1 Main handling of the user request

All the microservices and their interactions for fulfilling a user request to add an item to their shopping cart are shown in figure 1.6. The request to add an item to the shopping cart is divided into smaller tasks, each of which is handled by a separate microservice:

- The API Gateway microservice is responsible only for a cursory validation of the incoming request. Once it's validated, the work is delegated first to the Shopping Cart microservice and then to the Price Calculation microservice.
- The Shopping Cart microservice uses another microservice—the Product Catalog microservice—to look up the necessary information about the item being added to the cart. Shopping Cart then stores the user's shopping-cart information in its own data store and returns a representation of the updated shopping cart to API Gateway. For performance and robustness reasons, Shopping Cart will likely cache the responses from the Product Catalog microservice.
- The Price Calculation microservice uses the current business rules of the e-commerce website to calculate the total price of the items in the user's shopping cart, taking into account any applicable discounts.

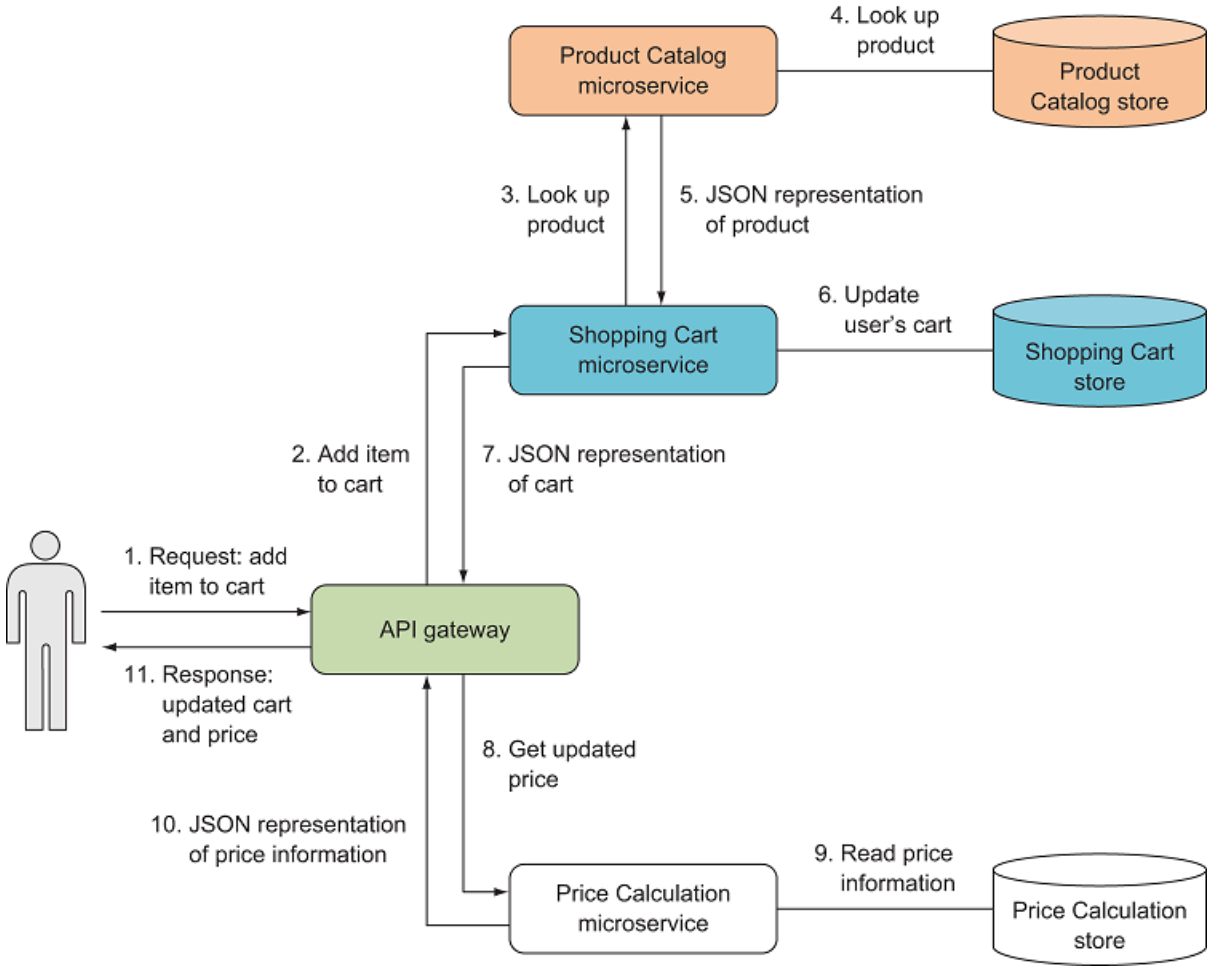

Figure 1.6 The API Gateway is all the client sees, but it's a thin layer in front of a system of microservices. The arrows indicate calls between different parts of the system, and the numbers on the arrows shows the sequence of calls.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Each of the microservices collaborating to fulfill the user's request has a single, narrowly focused capability and knows as little as possible about the other microservices. For example, the Shopping Cart microservice knows nothing about pricing or the Price Calculation microservice, and it knows nothing about how products are stored in the Product Catalog microservice. This is at the core of microservices: each one has a single responsibility.

1.7.2 Side effects of the user request

At this particular e-commerce website, when a user adds an item to their shopping cart, a couple of actions happen in addition to adding the item to the cart:

1. The recommendation engine updates its internal model to reflect the fact that the user has shown a high degree of interest in that particular product.
2. The tracking service records that the user added the item to their cart in the tracking database. This information may be used later for reporting or other business intelligence purposes.

Neither of these actions need to happen in the context of the user's request; they may as well happen after the request has ended, when the user has received a response and is no longer waiting for the back-end system.

You can think of these types of actions as side effects of the user's request. They aren't direct effects of the request to update the user's shopping cart; they're secondary effects that happen because the item was added to the cart. Figure 1.7 zooms in on the side effects of adding an item to the cart.

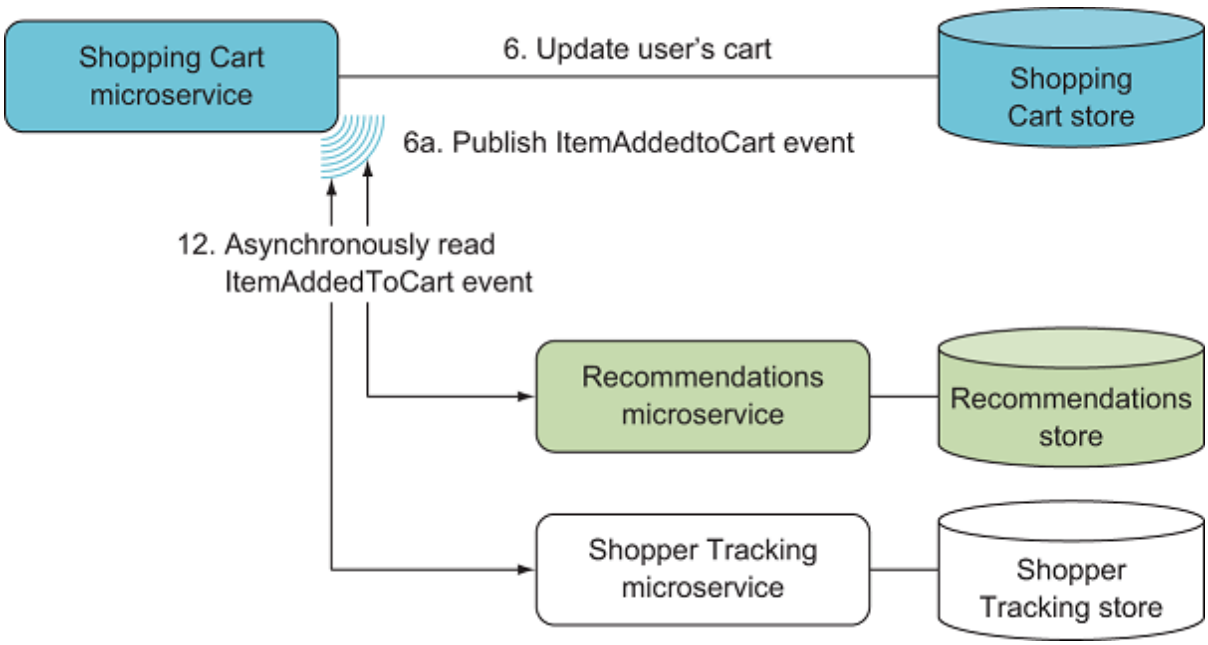

Figure 1.7 The Shopping Cart microservice publishes events, and other subscribing microservices react.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

The trigger for these side effects is an `ItemAddedToShoppingCart` event published by the Shopping Cart microservice. Two other microservices subscribe to events from Shopping Cart and take the necessary actions as events (such as `ItemAddedToShoppingCart` events) occur. These two subscribers react to the events asynchronously—outside the context of the original request—so the side effects may happen in parallel with the main handling of the request or after the main handling has completed.

> **NOTE** Implementing this type of event driven architecture using event feeds will be covered in chapter 5.

1.7.3 The complete picture

In total, six different microservices are involved in handling the request to add an item to a shopping cart, as shown in figure 1.8. None of these microservices know anything about the internals of the others. Five have their own private data stores dedicated to serving only their purposes. Some of the handling happens synchronously in the context of the user request, and some happens asynchronously.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

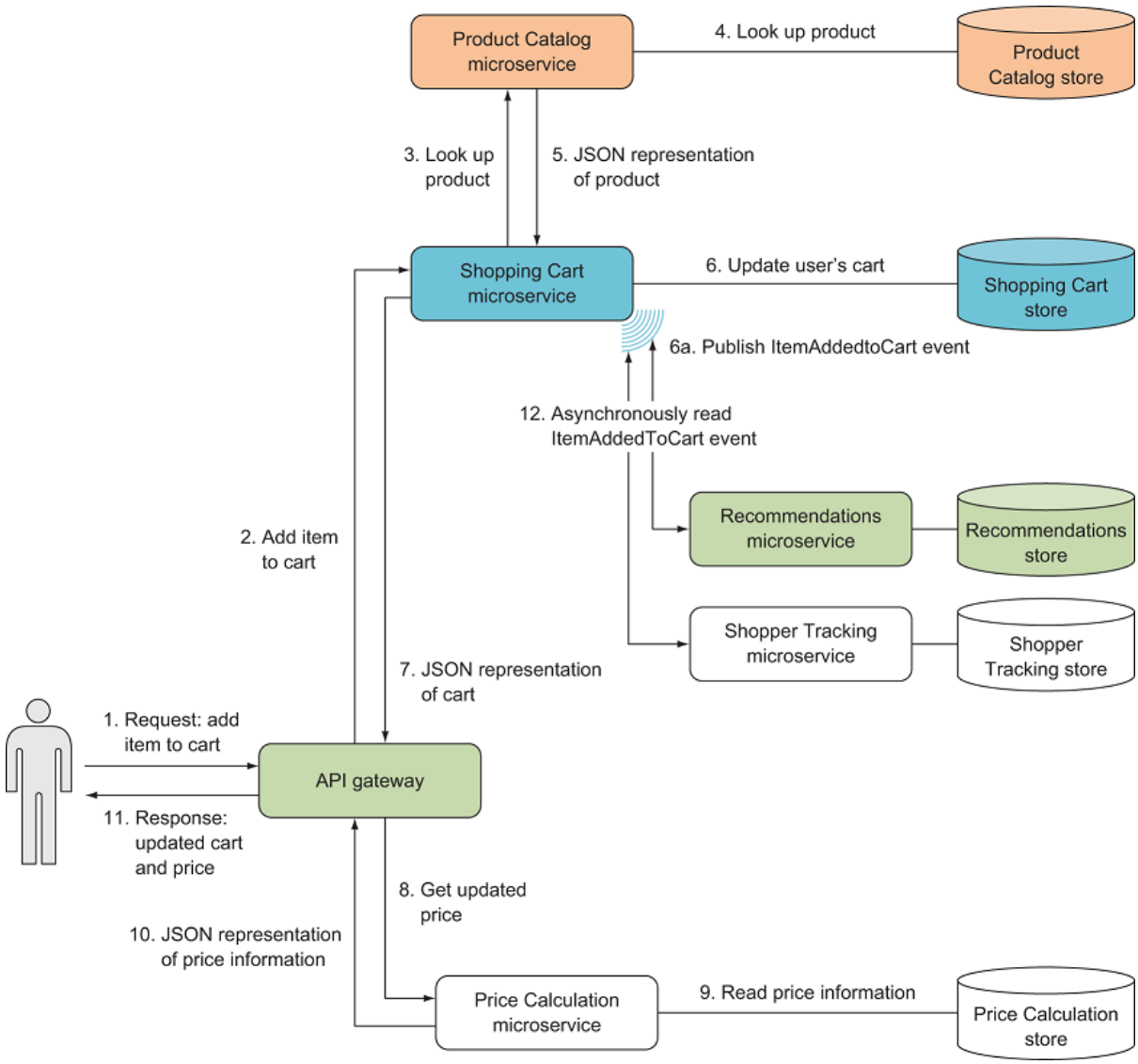

Figure 1.8 When a user adds an item to their shopping cart, the front end makes a request to the API Gateway microservice, which collaborates with other microservices to fulfill the request. During processing, microservices may raise events that other microservices can subscribe to and handle asynchronously.

This is a typical microservice system. Requests are handled through the collaboration of several microservices, each with a single responsibility and each as independent of the others as possible.

Now that we've taken a high-level look at a concrete example of how a microservices system can handle a user request, it's time to take a brief look at a .NET-based technology stack for microservices.

1.8 A .NET microservices technology stack

IIt's time to say hello to the technologies used most in this book: ASP.NET Core, MVC Core, and Kubernetes.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

1.8.1 ASP.NET Core and MVC Core

ASP.NET Core is Microsofts web platform which brings a number of benefits relevant to the microservices we will create throughout this book:

- *Cross platform*: ASP.NET Core is based on .NET 5 which works across Linux, Window, and more. This allows for some nice flexibilty when it comes to deciding where to run our microservices. We will be running them in Linux container on a Kubernetes cluster, but our microservice could also run the on a different container orchestrator, in Windows containers, on Windows servers or on Linux servers.
- *Fast*: ASP.NET Core is fast, and so is the underlying .NET 5 platform[8]. This gives a good foundation to build fast microservices.
- *Microservices are first class citizens*: One of the primary targets for ASP.NET Core is microservices, which means that the platform already implements some of the important plumbing that we would otherwise have to implement ourselves, such as request logging and monitoring support.
- *The middleware pipeline*: Middleware is lightweight, composable, and flexible way to build up a pipeline for handling incoming HTTP requests. Our microservices will have MVC Core at the end of their middleware pipelines.
- *Testable*: We will be taking advantage of the *TestHost* test helper for ASP.NET Core to test our microservices thoroughly in a style that lends itself to outside-in test-driven development.

On top of the ASP.NET Core platform we will use Microsofts MVC Core web framework, which provides us with an extra level of convenience when it comes to implementing handlers for HTTP endpoints.

1.8.2 Kubernetes

Kubernetes is a container orchestrator which we will use to run our microservices. Kubernetes allows us to automate deployment and to a certain degree the management of our microservices. In chapter 3 we will create a Kubernetes cluster in Azure and see how we can deploy our microservices to the Kubernetes cluster. In the remainder of the book following chapter 3 we will continue to deploy microservices to Kubernetes.

1.8.3 Setting up a development environment

Before you can start coding your first microservice, you need to have the right tools. To follow along with the examples in this book, you'll need four primary tools: An IDE, the .NET core command line, an HTTP tool, and Docker.

First you need a development environment for creating ASP.NET Core applications. The three most common options are:

- *Visual Studio*: Visual Studio is fully fledged IDE with great support for writing and debugging ASP.NET Core application. Visual Studio is Windows only and is probably the most widely used IDE for ASP.NET Core development.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

- *Visual Studio Code*: Visual Studio Code is a glorified editor that comes with a C# plugin that makes a very capable editor for writing ASP.NET Core applications. Visual Studio Code is lightweight and works across Windows, Linux, and Mac.
- *JetBrains Rider*: Rider is another fully fledged IDE with very strong support for writing and debugging ASP.NET Core applications. Rider works on Windows, Linux, and Mac alike.

All of these support developing, running, and debugging ASP.NET Core applications. They all give you a nice C# editor with IntelliSense and refactoring support. They're all aware of ASP.NET Core, and they can all launch and debug ASP.NET Core applications.

Second you need to have a version of the .NET Core command-line tool. If you did not get it bundled with the IDE, follow the instructions for installing .NET Core at http://dot.net. This gives you a command-line tool called `dotnet` that you'll use to perform a number of different tasks involved with microservices, including restoring NuGet packages, building, creating NuGet packages, and running microservices. Throughout the book I will use the `dotnet` command line for these tasks, but if you prefer you can also do most of them in your IDE.

Third you'll also need a tool for making HTTP requests. I recommend the REST Client plugin for Visual Studio Code, but Fiddler, Postman, and curl are also good and popular tools. You can use any of these to follow along with the examples in this book.

Fourth you need Docker installed and you need to have Kubernetes turned on in your Docker settings.

> **NOTE** In appendix A, you'll find download, installation, and quick usage information for Visual Studio, Visual Studio Code, JetBrains Rider, and Docker. Now is the time to set up the tools of your choice—you'll need them throughout the book.

1.9 A simple microservices example

Once you have a development environment up and running, it's time for a "Hello World" style microservices example. You'll use ASP.NET Core to create a microservice that has only a single API endpoint. Typically, a microservice has more than one endpoint, but one is enough for this example. The endpoint responds with the current UTC date and time in either JSON or XML format, depending on the request headers. This is illustrated in figure 1.9.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

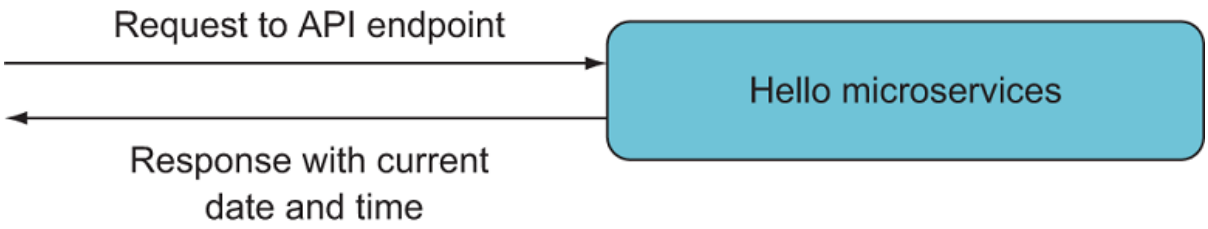

Figure 1.9 A "Hello World"-style microservice that responds with the current date and time

> **NOTE** When I talk about an *API endpoint*, an *HTTP endpoint*, or just an *endpoint*, I mean a URL where one of your microservices reacts to HTTP requests.

To implement this example, you'll follow these three steps:

1. Create an empty ASP.NET Core application.
2. Add MVC Core to the application.
3. Add a MVC controller with an implementation of the endpoint.

The following sections will go through each step in detail.

1.9.1 Creating an empty ASP.NET Core application

The first thing you need to do is create an empty ASP.NET Core application called *HelloMicroservices* using the .NET Core command line tool. If you prefer, you can create the project using your IDE - the result is the same. To use the .NET Core command line tool open up a shell and issue the command `dotnet new web -n HelloMicroservices`.

Once you've created your empty ASP.NET Core application and named it HelloMicroservices, you should have a folder called `HelloMicroservices` that contains these files:

- appsettings.Development.json
- appsettings.json
- HelloMicroservices.csproj
- Program.cs
- Startup.cs

There are other files in the project, but these are the one you'll be concerned with right now.

This is a complete application, ready to run. It will respond to HTTP GET requests to the path `/` with the string "Hello World". You can start the application from the command line by going to the `HelloMicroservices` folder and typing the command `dotnet run`.

The application runs on localhost port 5000 (note that if you choose to run it from inside Visual Studio, you may get another port). If you go to http://localhost:5000 in a browser, you'll get the "Hello World" response.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

1.9.2 Adding MVC Core to the project

To add MVC Core to the project we will change the `startup` class. There are two parts to adding MVC to our microservice:

1. Adding the necessary types to the *service collection*. ASP.NET Core is designed to use *dependency injection* [9] and all types that ASP.NET Core should be able to instantiate must be added to the service collection.
2. Adding all the endpoints we will write in our microservices to ASP.NET Core's *routing table*. The routing table is used by ASP.NET Core every time an HTTP request comes in to try to find an endpoint that matches the request.

The `startup.cs` file is changed so it looks like this:

Listing 1.1 Startup with MVC

```
 language="cs" linenumbering="unnumbered">namespace HelloMicroservices
{
  using Microsoft.AspNetCore.Builder;
  using Microsoft.AspNetCore.Hosting;
  using Microsoft.Extensions.DependencyInjection;

  public class Startup
  {
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddControllers();        ❶
    }

    public void Configure(
      IApplicationBuilder app,
      IWebHostEnvironment env)
    {
      app.UseHttpsRedirection();        ❷
      app.UseRouting();
      app.UseEndpoints(endpoints =>
        endpoints.MapControllers());    ❸
    }
  }
}
```

❶ Adds MVC controller and helper services to the service collection

❷ Redirects all HTTP requests to HTTPS

❸ Adds all endpoints in all controllers to MVCs route table

Apart from adding controller and helpers to the services collection and adding our endpoints to routing table we also added a line that redirects all HTTP requests to HTTPS. Our application will handle HTTPS requests on port 5001. For example s request to http://localhost:5000/ will be redirected to https://localhost:5001/. The first time you attempt to make a request to https://localhost:5001/ you will probably be warned about an unknown certificate. This is a development certificate that your microservices will use localhost and you can safely accept the certificate.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

At this point, you have an ASP.NET Core application with MVC added; but the application can't handle any requests yet, because you haven't set up any endpoints. If you restart the application and again go to https://localhost:5001 in a browser or in Visual Studio Codes REST client plugin, you'll get a "404 page not found" response. Let's fix that.

1.9.3 Adding an MVC controller with an implementation of the endpoint

Now you'll add an MVC controller with an implementation of the single API endpoint. We will implement controllers as classes that inherit from `Controller` and we will use them to declare endpoints and to implement the behavior for each endpoint. Because we added the `endpoints.MapControllers` line to `Startup` in the previous section ASP.NET Core will discover the endpoints we declare in our controllers. We add a controller by creating a file called `CurrentDateTimeController.cs` and adding the following code to it.

Listing 1.2 MVC Controller

```
 language="cs" linenumbering="unnumbered">namespace HelloMicroservices
{
  using System;
  using Microsoft.AspNetCore.Mvc;

  public class CurrentDateTimeController : Controller ❶
  {
    [HttpGet("/")]                                    ❷
    public object Get() => DateTime.UtcNow;           ❸
  }
}
```

❶ Declares an MVC Controller

❷ Declares an HTTP GET endpoint the the path /

❸ Return the current date and time as response to requests

In this controller, you declare a route for the path / with the with the attribute `HttpGet("/")`. This tells MVC that any HTTP GET request to / should be handled by the method following the attribute, the method `Get()` in this case. The `Get()` method returns the current UTC date and time, so that is the response to every HTTP GET request to /.

You can now rerun the application and again point your browser to https://localhost:5001. Your browser will hit the route on your MVC Controller and show current UTC date and time as a string. We can do the same with the REST client plugin in Visual Studio Code which looks like figure 1.10.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

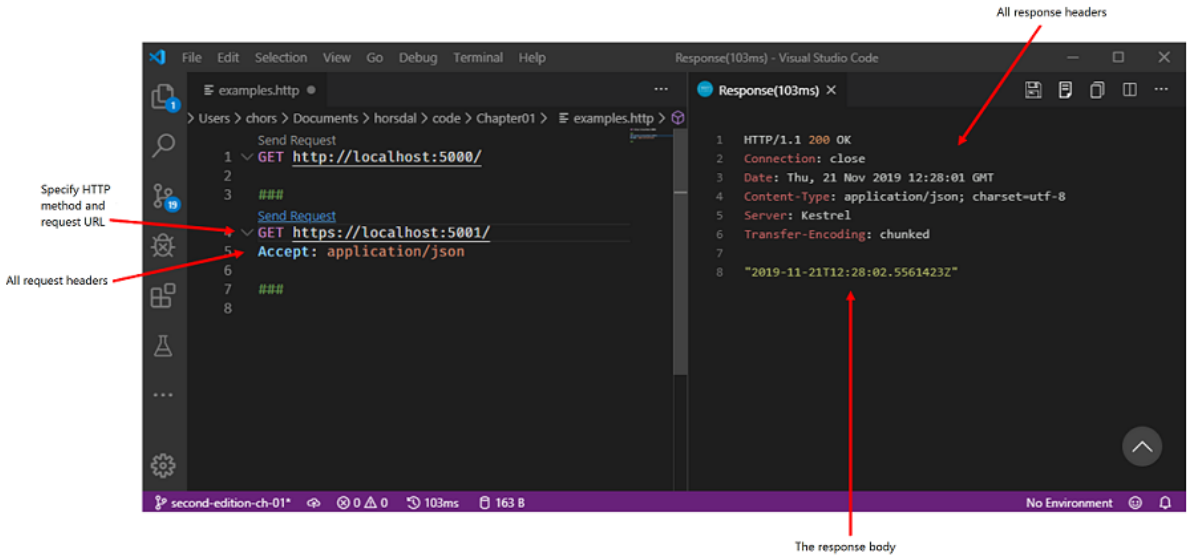

Figure 1.10 Restclient in VS Code makes it easy to send HTTP requests and control the request details, such as the headers and HTTP method.

On the wire, this is the request:

GET / HTTP/1.1 Host: localhost:5001 Accept: application/json

The response from this request is the current UTC data and time:

HTTP/1.1 200 OK Connection: close Date: Thu, 21 Nov 2019 12:28:01 GMT Content-Type: applicatic charset=utf-8 Server: Kestrel Transfer-Encoding: chunked "2019-11-21T12:28:02.5561423Z"

You've now implemented your first HTTP endpoint with MVC Core.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

1.10 Summary

- Microservices is an overloaded term used both for the microservices architectural style and for individual microservices in a system of microservices.
- The microservices architectural style is a special form of SOA, where each service is small and provides one and only one business capability.
- A microservice is a service with a single narrowly focused capability.
- I'll refer to six characteristics of a microservice in this book. A microservice
  - Is responsible for providing a single capability.
  - Is individually deployable. You must be able to deploy every microservice on its own without touching any other part of the system.
  - Runs in one or more processes, separate from other microservices.
  - Owns and stores the data belonging to the capability it provides in a data store that the microservice itself has access to.
  - Is small enough that a small team of around five people can develop and maintain a few handfuls or more of them.
  - Replaceable. The team should be able to rewrite a microservice from scratch in a short period of time if, for instance, the codebase has become a mess.
- Microservices go hand in hand with continuous delivery:
  - Having small, individually deployable microservices makes continuous delivery easier.
  - Being able to deploy automatically, quickly, and reliably simplifies deploying and maintaining a system of microservices.
- A system built with microservices allows for scalability and resilience.
- A system built with microservices is malleable: it can be easily changed according to your business needs. Each microservice by itself is highly maintainable, and even creating new microservices to provide new capabilities can be done quickly.
- Microservices collaborate to provide functionality to the end user.
- A microservice exposes a remote public API that other microservices may use.
- A microservice can expose a feed of events that other microservices can subscribe to. Events are handled asynchronously in the subscribers but still allow subscribers to react to events quickly.
- ASP.NET Core is well suited for implementing microservices
- Kubernetes is the container orchestrator that we will use throughout the book
- Most microservices don't serve HTML from their endpoints, but rather data in the form of JSON or XML. Applications like RESTClient and Fiddler are good for testing such endpoints.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

2

A basic shopping cart microservice

This chapter covers

- A nearly complete implementation of the Shopping Cart microservice
- Creating HTTP endpoints with MVC Core
- Implementing a request from one microservice to another
- Implementing a simple event feed for a microservice

In chapter 1, we looked at how microservices work and how they can be characterized. You also set up a simple technology stack—C#/MVC Core/ASP.NET Core—that lets you create microservices easily, and you saw a basic Shopping Cart microservice. In this chapter, you'll implement the four main parts of this microservice using the same technology stack:

- A basic HTTP-based API allowing clients to retrieve a cart, delete it, and add items to it. Each of these methods will be visible as an HTTP endpoint, such as http://myservice/add/{item_number}.
- A call from one service to another for more information. In this case, the Shopping Cart microservice will ask the Product Catalog microservice for pricing information based on the `item_number` of the item being added to the cart.
- An event feed that the service will use to publish events to the rest of the system. By creating an event feed for the shopping cart, you'll make it possible for other services (such as a recommendation engine) to update their own data and improve their capabilities.
- The domain logic for implementing the behavior of the shopping cart.

To keep things simple, you won't do a complete implementation of this microservice in this chapter. We'll look at the following topics and complete the microservice during the course of the book:

- The Shopping Cart microservice should have its own data store, but you won't

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

implement it or the data access code to get data in and out of it. Chapter 6 covers this in full.

- Any production-ready microservice should include support for monitoring and logging. If a microservice doesn't provide regular insight into its health, it becomes difficult to keep the overall system running steadily. But these functions don't directly provide a business capability, so I've left logging and monitoring capabilities to be discussed in chapter 10.

Let's get to it.

> **NOTE** Be sure you've set up your development environment. In appendix A, you'll find download, installation, and quick usage information about IDEs you can use to follow along with the code throughout this book. This chapter has lots of code, so if you haven't already set up a development environment, now is the time to do it.

2.1 Overview of the Shopping Cart microservice

In chapter 1, we looked at how an e-commerce site built with microservices might handle a user's request to add an item to their shopping cart. The complete overview of how the request is handled is repeated in figure 2.1

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

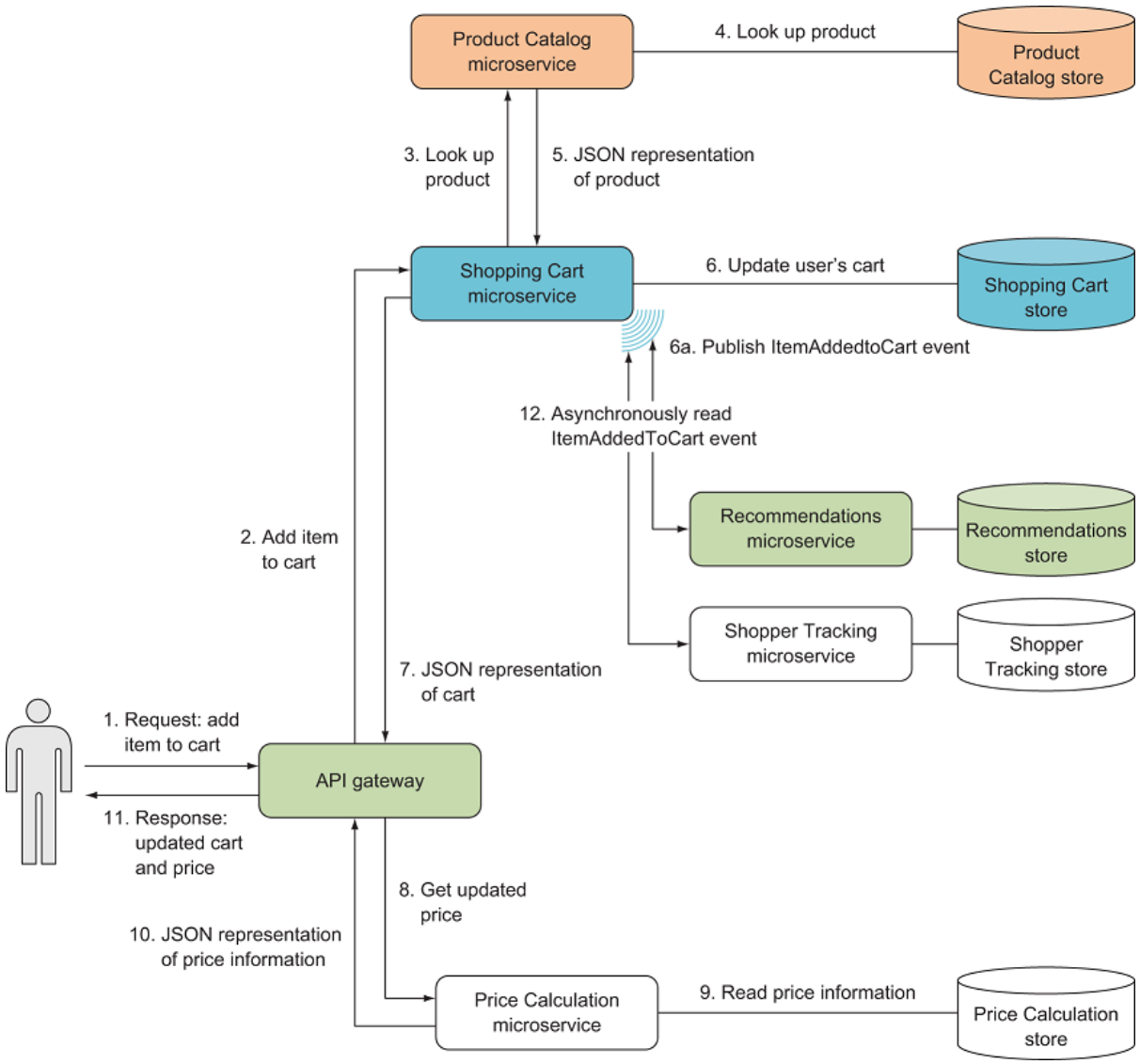

Figure 2.1 The Shopping Cart microservice allows other microservices to get a shopping cart, add items to and delete items from a shopping cart, and subscribe to events from Shopping Cart.

The Shopping Cart microservice plays a central role when a user wants to add an item to their shopping cart. But it's not the only process in which Shopping Cart plays a role. It's equally important to let the user see their shopping cart and delete an item from it. The Shopping Cart microservice must support those processes through its HTTP API, just as it supports adding an item to a shopping cart. Figure 2.2 shows the interactions between the Shopping Cart microservice and the other microservices in the system.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

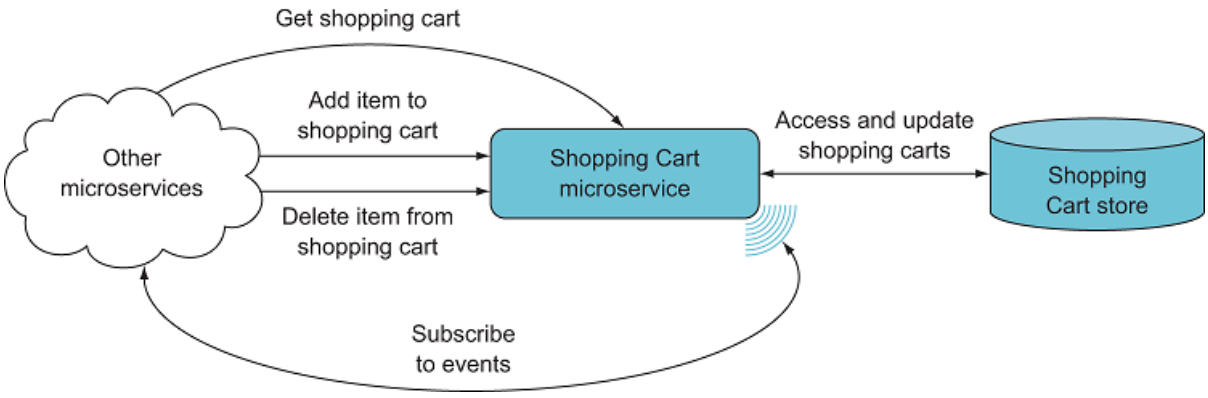

Figure 2.2 Overview of how an e-commerce site built with microservices can handle adding an item to a user's shopping cart

The Shopping Cart microservice supports three types of synchronous requests:

- Getting a shopping cart
- Adding an item to a shopping cart
- Deleting an item from a shopping cart

On top of that, it exposes an event feed that other microservices can subscribe to. Now that you've seen an overview of the Shopping Cart microservice's complete functionality, you can start drilling into its implementation.

2.1.1 Components of the Shopping Cart microservice

Let's zoom in and see what this microservice looks like at closer range. As shown in figure 2.3, the Shopping Cart microservice consists of these components:

- A small Shopping Cart domain model that's responsible for implementing any business rules related to shopping carts.
- An HTTP API component that's responsible for handling all incoming HTTP requests. The HTTP API component is divided into two parts: one handles requests from other microservices to do something, and the other exposes an event feed.
- Two data store components: `EventStore` and `ShoppingCartStore`. These data store components are responsible for talking to the data store (`ShoppingCartStore`):
  - EventStore handles saving events to and reading them from the data store.
  - ShoppingCartStore handles reading and updating shopping carts in the data store. Note that shopping carts and events may be stored in different databases; we'll return to this in chapter 6.
- A `ProductCatalogClient` component that's responsible for communicating with the Product Catalog microservice shown in figure 2.1. Placing that communication in `ProductCatalogClient` serves several purposes:
  - It encapsulates knowledge of the other microservice's API in one place.
  - It encapsulates the details of making an HTTP request.
  - It encapsulates caching results from the other microservice.
  - It encapsulates handling errors from the other microservice.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

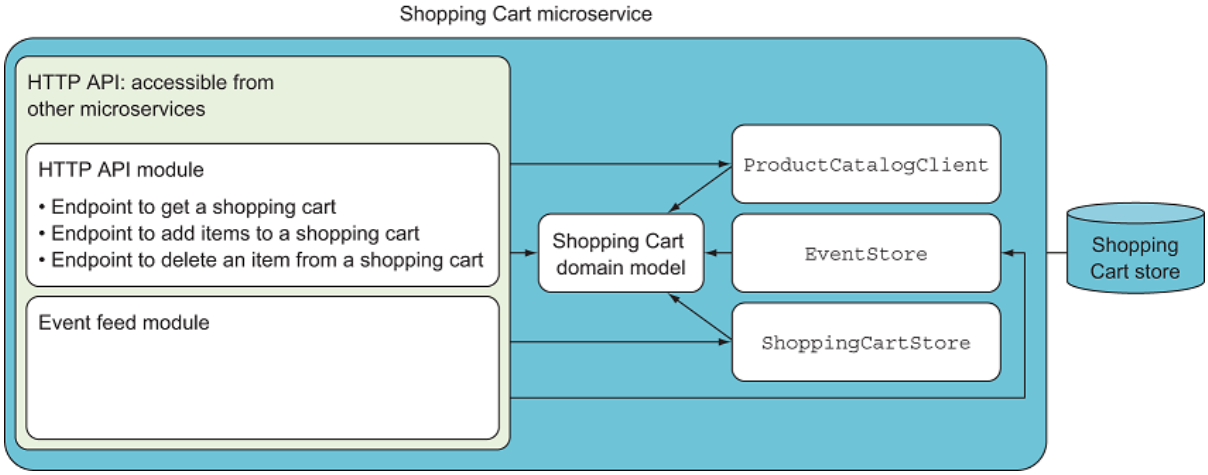

Figure 2.3 The Shopping Cart microservice is a small codebase with a few components that provide one focused business capability.

This chapter includes the code for the domain model, the HTTP API, and a basic implementation of `ProductCatalogClient`, but skips `EventStore` and `ShoppingCartStore` and the data store. In addition, for the sake of brevity, this chapter omits error-handling code. Chapter 5 will go further into detail about how to implement microservice APIs easily with ASP.NET Core; chapter 6 will also return to the subject of storing data in a microservice. Chapter 7 will dive deeper into how to design robustness into clients such as `ProductCatalogClient`.

2.2 Implementing the Shopping Cart microservice

Now that you understand the Shopping Cart microservice's components, it's time to get into the code.

SIDEBAR **New technologies used in this chapter**

In this chapter, you'll begin using two new technologies:

- `HttpClient` is a .NET Core type for making HTTP requests. It provides an API for creating and sending HTTP requests as well as reading the responses that come back.
- Polly is a library that makes it easy to implement the more common policies for handling remote-call failures. Out of the box, Polly has support for various retry and circuit breaker policies. I'll discuss circuit breakers in chapter 7.
- Scrutor is a library that add a number of convenient extensions to ASP.NET Core's built-in dependency injection container

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

2.2.1 Creating an empty project

The first thing you need to do is set up an ASP.NET Core project, just as in chapter 1. First, create an empty ASP.NET Core application called `ShoppingCart` - on the command line you can use the command `dotnet new web -n ShoppingCart`. Second, add MVC Core to the application in the `Startup` class:

Listing 2.1 `Startup` class with MVC

```
 language="sql" linenumbering="unnumbered">namespace ShoppingCart
{
  using Microsoft.AspNetCore.Builder;
  using Microsoft.AspNetCore.Hosting;
  using Microsoft.Extensions.DependencyInjection;

  public class Startup
  {
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddControllers();        ❶
    }

    public void Configure(IApplicationBuilder app)
    {
      app.UseHttpsRedirection();        ❷
      app.UseRouting();
      app.UseEndpoints(endpoints =>
        endpoints.MapControllers());    ❸
    }
  }
}
```

❶ Adds MVC controller and helper services to the service collection

❷ Redirects all HTTP requests to HTTPS

❸ Adds all endpoints in all controllers to MVCs route table

You now have an empty application that's ready to go.

2.2.2 The Shopping Cart microservice's API for other services

In this section, you'll implement the Shopping Cart microservice's HTTP API, which is highlighted in figure 2.4. This API has three parts, each of which is implemented as an HTTP endpoint:

- An HTTP `GET` endpoint where other microservices can fetch a user's shopping cart by providing a user ID. The response is a shopping cart serialized as either JSON or XML.
- An HTTP `POST` endpoint where other microservices can add items to a user's shopping cart. The items to be added are passed to the endpoint as an array of product IDs. The array can be in XML or JSON, and it must be the body of the request.
- An HTTP `DELETE` endpoint where other microservices can remove items from a user's shopping cart. The items to be deleted are passed in the body of the request as an XML or JSON array of product IDs.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

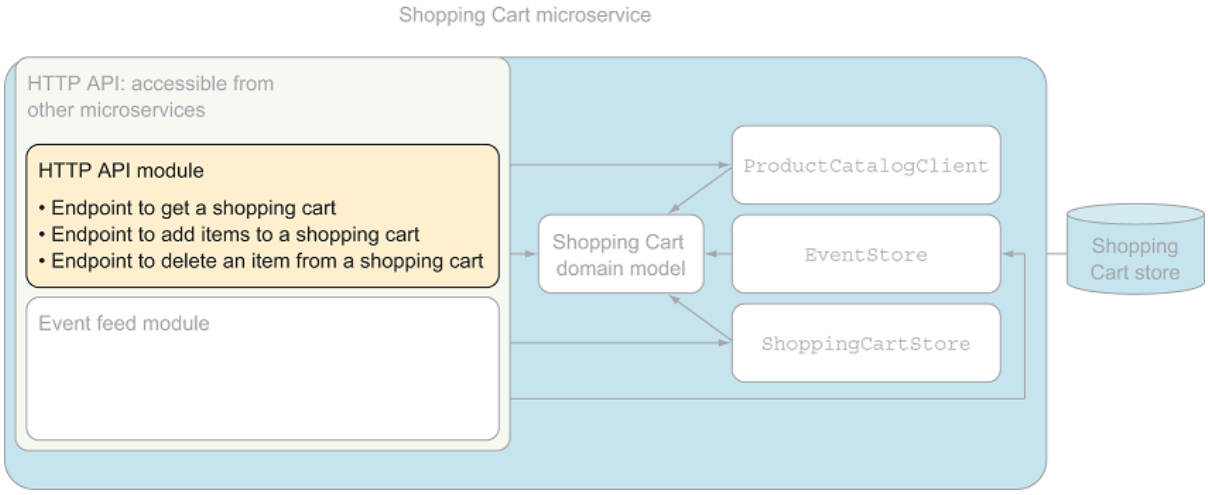

Figure 2.4 Implementing the HTTP API component

The following three sections each implement one of the endpoints.

NOTE **Note**

I will not show implementations of `EventStore` or `ShoppingCartStore`. Chapter 6 will show proper implementation of classes like `EventStore` and `ShoppingCartStore`, but for now a simple hard coded implementation is sufficient. You can get an implementation of `EventStore` and `ShoppingCartStore` from the books code package or you create one yourself.

GETTING A SHOPPING CART

The first part of the HTTP API that you'll implement is the endpoint that lets other microservices fetch a user's shopping cart. Figure 2.5 shows how other microservices can use an endpoint to get a shopping cart.

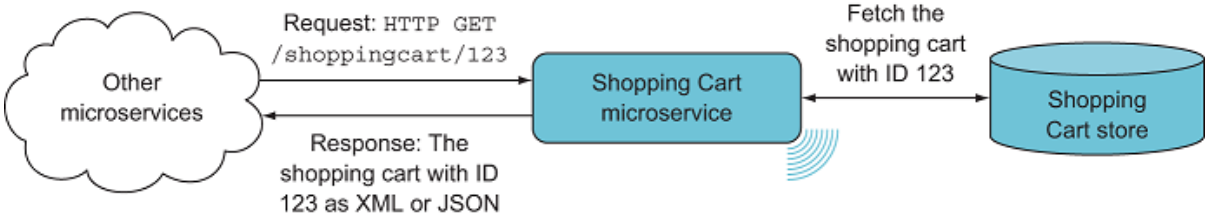

Figure 2.5 Other microservices can use an endpoint on Shopping Cart to get a shopping cart in XML or JSON format.

The endpoint accepts `HTTP GET` requests. Its URL includes the ID of the user whose shopping cart the other microservice wants, and the body of the response is an XML or JSON serialization of that shopping cart. The request should include an `Accept` header indicating whether the response body should be XML or JSON.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

For example, the API gateway in figure 2.1 may need the shopping cart for a user with ID `123`. To get that, it sends this HTTP request:

HTTP GET /shoppingcart/123 HTTP/1.1 Host: shoppingcart.my.company.com Accept: application/jso

This is a request to `shoppingcart/123` on the Shopping Cart microservice, and the `123` part of the URL is the user ID.

To handle such requests, you need to add a controller to the `ShoppingCart` project called `ShoppingCartController`. As mentioned in chapter 1, we will implement controllers by inheriting from `Controller` and we will use controllers to implement HTTP endpoints. Put the following code in a new file called ShoppingCartController.cs.

Listing 2.2 Endpoint to access a shopping cart by user ID

```
 language="sql" linenumbering="unnumbered">namespace ShoppingCart.Shoppingcart
{
  using Microsoft.AspNetCore.Mvc;
  using ShoppingCart;

  [Route("/shoppingcart")]                          ❶
  public class ShoppingCartController : Controller  ❷
  {
    private readonly IShoppingCartStore shoppingCartStore;

    public ShoppingCartController(IShoppingCartStore shoppingCartStore)
    {
      this.shoppingCartStore = shoppingCartStore;
    }

    [HttpGet("{userId:int}")]                       ❸
    public ShoppingCart Get(int userId) =>          ❹
      this.shoppingCartStore.Get(userId);           ❺
  }
}
```

❶ Tells MVC that all routes in this controller start with `/shoppingcart`

❷ Declares `ShoppingCartController` as a controller

❸ Declares the endpoint for handling requests to `/shoppingcart/{userid}`, such as `/shoppingcart/123`

❹ Assigns the `{userId}` from the URL to the `userId` variable

❺ Returns the user's shopping cart. MVC serializes it before sending it to the client.

Let's break down this code.

The attribute `[HttpGet("{userId:int}")]` declares that you want to handle HTTP `GET` requests to endpoints matching the pattern inside the parenthesis. The pattern can be a literal string, like `"/shoppingcart";` or it can contain segments that match and capture parts of the request URL, like `{userid:int}` or it combine literal strings and segments, like

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

`"/shoppingcart/{userId:int}"`. The `{userId:int}` is called `userId` and is constrained to only match integers.

The `HttpGet` attribute is followed by a method:

public ShoppingCart Get(int userId) => this.shoppingCartStore.Get(userId);

This is the action method, and it's the piece of code that's executed every time the Shopping Cart microservice receives a request to a URL that matches the route declaration. For instance, when the API gateway requests a shopping cart via the URL path `/shoppingcart/123`, this is the method that handles the request.

The action method can take arguments which match segments of the URL. For instance the `int userId` is matched to the `{userId}` segment based on the names of the argument and the segment.

The action method uses a `shoppingCartStore` object that the `ShoppingCartController` constructor takes as an argument and assigned to a instance variable:

public ShoppingCartController(IShoppingCartStore shoppingCartStore) { this.shoppingCartStore = shoppingCartStore; }

The constructor argument has the type `IShoppingCartStore`, which is an interface. If an implementation of `IShoppingCartStore` is registered in ASP.NET Core's service collection an instance will be given to the `ShoppingCartController` constructor each time the framework creates an instance of the controller. We will add code to the `Startup` class that registers an implementation of `IShoppingCartStore`. Instead of explicitly registering the `ShoppingCartStore` class as the implementation `IShoppingCartStore` we will add code that scans the shopping cart project and adds all classes that implement interfaces to the service collection. To that end we will use the NuGet package Scrutor which is a collection of convenient extensions to `IServiceCollection`. So first we add Scrutor to the shopping cart project by running this dotnet command:

PS> dotnet add package scrutor

This installs the Scrutor NuGet package which we can see in the shopping cart project file - `ShoppingCart.csproj` - where there now is a package reference to Scrutor:

<Project Sdk="Microsoft.NET.Sdk.Web"> <PropertyGroup> <TargetFramework>netcoreapp3.0</TargetFramework> </PropertyGroup> <ItemGroup> <PackageRe Include="Scrutor" Version="3.1.0" /> </ItemGroup> </Project>

With Scrutor installed we can add this code to the `Startup` class:

services.Scan(selector => selector .FromAssemblyOf<Startup>() .AddClasses() .AsImplementedInterfa

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

If there is a class implementing the `IShoppingCartStore` interface this code will be register it in the service collection and ASP.NET Core will be able to inject it into the `ShoppingCartController` constructor. I'm leaving out the data-storage code in this chapter, but the code in the code download accompanying this book contains the `IShoppingCart` interface and a dummy implementation of it.

The action method returns a `ShoppingCart` object that it gets back from `shoppingCartStore`:

this.shoppingCartStore.Get(userId);

The `ShoppingCart` type is specific to the Shopping Cart microservice, so MVC has no way of knowing about this particular type. But you can return any object you want, and MVC will serialize it and return the data to the caller.

The following listing shows an example of the response to a request to /shoppingcart/123.

Listing 2.3 Example response from the Shopping Cart microservice

```
 language="sql" linenumbering="unnumbered">HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8           ❶

539                                                     ❷
{                                                       ❸
    "userId": 42,
    "items": [
        {
            "productcatalogId": 1,
            "productName": "Basic t-shirt",
            "description": "a quiet t-shirt",
            "price": {
                "currency": "eur",
                "amount": 40
            }
        },
        {
            "productcatalogId": 2,
            "productName": "Fancy shirt",
            "description": "a loud t-shirt",
            "price": {
                "currency": "eur",
                "amount": 50
            }
        }
    ]
}
```

❶ The response body is in JSON.

❷ Length of the response body

❸ Shopping cart serialized as JSON

As you can see, you can get a lot of functionality up and running with a small amount of code.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

ADDING ITEMS TO A SHOPPING CART

The second endpoint you need to add to the Shopping Cart microservice lets you add items to a user's shopping cart. Figure |2.6 shows how other microservices can use this endpoint.

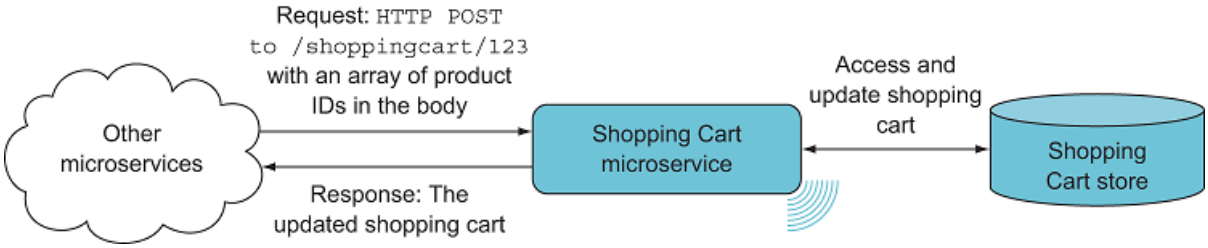

Figure 2.6 Other microservices can add items to a shopping cart with an HTTP POST request that includes an array of product IDs in the request body.

Like the HTTP `GET` endpoint in the previous section, this new endpoint receives a user ID in the URL. This time, the endpoint accepts HTTP `POST` requests instead of HTTP `GET`, and the request should provide a list of items in the body of the request. For example, the following request adds two items to user 123's shopping cart.

Listing 2.4 Adding two items to a shopping cart

```
language="sql" linenumbering="unnumbered">  POST /shoppingcart/123/items HTTP/1.1       ❶
 Host: shoppingcart.my.company.com
 Accept: application/json                    ❷
 Content-Type: application/json              ❸

 [1, 2]                                      ❹
```

❶ The URL includes the ID of the shopping cart: 123.

❷ The response should be in JSON format.

❸ The data in the request body is in JSON.

❹ The request body is a JSON array of product IDs.

To handle such requests, you need to add another action method to `ShoppingCartController`. The new action method reads the items from the body of the request, looks up the product information for each one, adds them to the correct shopping cart, and returns the updated shopping cart. The code to fetch product information is shown in section 2.2.3

The new action method is shown in the next listing. Add it to the `ShoppingCartController`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 2.5 Handler for a route to add items to a shopping cart

```
 language="sql" linenumbering="unnumbered">[Route("/shoppingcart")]
public class ShoppingCartController : Controller
{
  private readonly IShoppingCartStore shoppingCartStore;
  private readonly IProductCatalogClient productCatalogClient;
  private readonly IEventStore eventStore;

  public ShoppingCartController(
    IShoppingCartStore shoppingCartStore,
    IProductCatalogClient productCatalogClient,
    IEventStore eventStore)
  {
    this.shoppingCartStore = shoppingCartStore;
    this.productCatalogClient = productCatalogClient;
    this.eventStore = eventStore;
  }

  [HttpGet("{userId:int}")]
  public ShoppingCart Get(int userId) =>
    this.shoppingCartStore.Get(userId);

  [HttpPost("{userId:int}/items")]                          ❶
  public async Task<ShoppingCart> Post(
    int userId,
    [FromBody] int[] productIds)                            ❷
  {
    var shoppingCart = shoppingCartStore.Get(userId);
    var shoppingCartItems =
      await this.productcatalog
        .GetShoppingCartItems(productIds);                  ❸
    shoppingCart.AddItems(shoppingCartItems, eventStore);   ❹
    shoppingCartStore.Save(shoppingCart);                   ❺
    return shoppingCart;                                    ❻
  }
}
```

❶ Declares an HTTP POST endpoint for /shoppingcart/{userid}/item

❷ Reads and deserializes the array of product IDs in the HTTP request body

❸ Fetches the product information from the Product Catalog microservice.

❹ Adds items to the cart

❺ Saves the updated cart to the data store

❻ Returns the updated cart

Two new MVC capabilities are at play here. First, the new action method is asynchronous. The action is declared asynchronous because it makes a remote call to the Product Catalog microservice. Performing that external call asynchronously saves resources in Shopping Cart. ASP.NET Core can run fully asynchronously, which allows application code to make good use of C#'s `async/await` feature.

Second, the body of the request contains a JSON array of product IDs. These are the items that should be added to the shopping cart. The action uses MVC model binding to read these into a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

C# array:

[FromBody] int[] productIds

Model binding supports any serializable C# object. You'd often use a more structured object than a flat JSON array to send data into an endpoint, and reading that would be just as easy as in this case. The type of the parameter `int[] productIds` would just need to be changed to a type other than `int[]`.

The new action handler uses two objects that aren't already present in `ShoppingCartController`. You once again rely on ASP.NET Core to provide them through constructor arguments.

Listing 2.6 Adding module dependencies as constructor arguments

```
language="sql" linenumbering="unnumbered">public ShoppingCartController(
 IShoppingCartStore shoppingCartStore,
 IProductCatalogClient productCatalogClient,
 IEventStore eventStore)                          (1)
```

(1) Only used to pass into the AddItems call, where it will be used later

Other microservices can now add items to shopping carts. They should similarly be allowed to remove items from shopping carts.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **async/await at a glance**

C# 5 introduced two new keywords, `async` and `await`, to allow methods to run asynchronously easily. A basic `async` method looks like this:

```
public async Task<int> WaitForANumber()  {
  await Task.Delay(1000)                 return 10;              }
```

When you call this method, the thread of execution continues as usual until `await`. The `await` keyword works in conjunction with *awaitables*—the most common await-able is `System.Threading.Tasks.Task<T>` —and asynchronously waits until the awaitable completes. This means two things happen when execution reaches `await`:

- The remainder of the method is queued up for execution when the await-able—in this case, the `Task` returned from `Task.Delay(1000)` —completes. When the awaitable completes, the rest of the method is executed, possibly on a new thread but with same state as before the `await` reestablished.
- The current thread of execution returns from the `async` method and continues in the caller.

In server-side code, like microservices, many requests require some I/O, such as calling a data store or another microservice. If you can execute the I/O asynchronously instead of blocking a thread while waiting for the I/O to complete, you save resources on your servers. In some situations, you may also gain some performance, but that isn't the general case. I use `async/await` and `Task` s a lot in this book to save resources on the server and gain scalability.

REMOVING ITEMS FROM A SHOPPING CART

The third and last endpoint is an HTTP `DELETE` endpoint that, as shown in figure 2.7, lets other microservices remove items from shopping carts. You should now have the hang of adding endpoints to controllers. You need to implement an HTTP `DELETE` endpoint that takes an array of product IDs and removes those products from the cart. Add the following code to the `ShoppingCartController`.

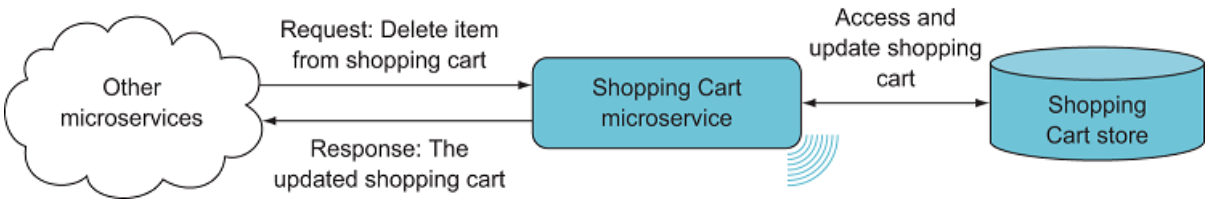

Figure 2.7 Other microservices can remove items from a shopping cart with an HTTP DELETE request by providing an array of product IDs in the request body.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 2.7 Endpoint for removing items from a shopping cart

```
 language="sql" linenumbering="unnumbered">[HttpDelete("{userid:int}/items")]    ❶
public ShoppingCart Delete(
  int userId,
  [FromBody] int[] productIds)
{
  var shoppingCart =
    this.shoppingCartStore.Get(userId);
  shoppingCart.RemoveItems(
    productIds,
    this.eventStore);                  ❷
  this.shoppingCartStore.Save(shoppingCart);
  return shoppingCart;
}
```

❶ Using the same route template for two route declarations is fine if they use different HTTP methods.

❷ The eventStore will be used later in the RemoveItems method.

This completes `ShoppingCartController`, which ends up at about than 50 lines of code. These are the kind of sizes we work with in microservice systems.

2.2.3 Fetching product information

Now that the API exposed by the Shopping Cart microservice is implemented, let's switch gears and look at how the product information is fetched from the Product Catalog microservice. Figure 2.8 highlights `ProductCatalogClient`, which you'll implement in this section.

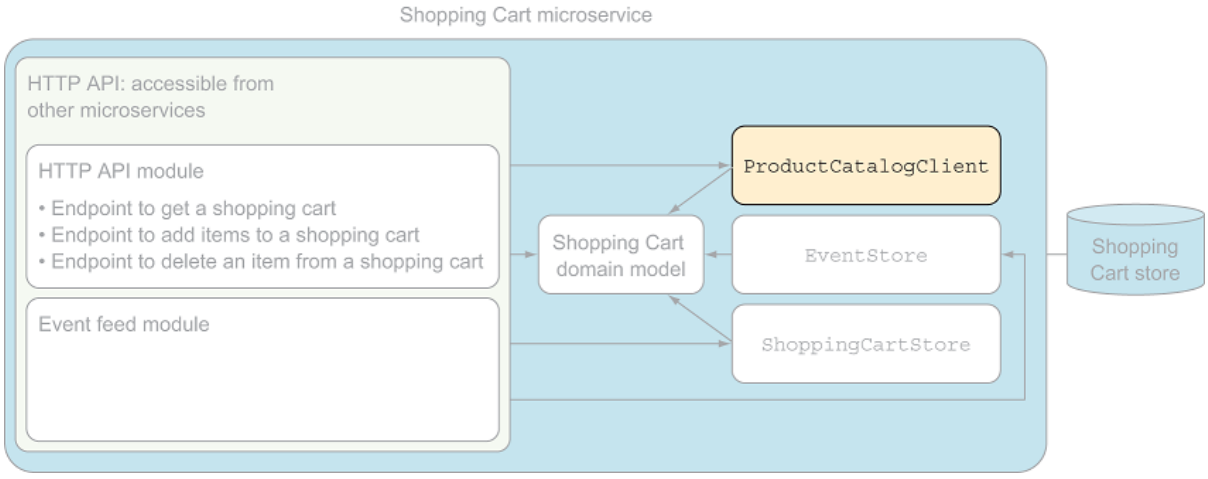

Figure 2.8 ProductCatalogClient

The Product Catalog microservice and the Shopping Cart microservice are separate microservices running in separate processes, perhaps even on separate servers. Product Catalog exposes an HTTP API that Shopping Cart uses. Product catalog information is fetched in HTTP `GET` requests to an endpoint on the Product Catalog microservice.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

You need to follow these three steps to implement the HTTP request to the Product Catalog microservice:

1. Implement the HTTP `GET` request.
2. Parse the response from the endpoint at the Product Catalog microservice, and translate it to the domain of the Shopping Cart microservice.
3. Implement a policy for handling failed requests to the Product Catalog microservice.

The subsequent sections walk you through these steps.

IMPLEMENTING THE HTTP GET

The Product Catalog microservice exposes an endpoint at the path /products. The endpoint accepts an array of product IDs as a query string parameter and returns the product information for each of those products. For example, the following request fetches the information for product IDs 1 and 2:
HTTP GET /products?productIds=[1,2] HTTP/1.1 Host: productcatalog.my.company.com Accept: app

You'll use the `HttpClient` type to perform the HTTP request. Instead of a real Product Catalog microservice, the implementation makes a call to a hardcoded json file on GitHub which serves as a fake version of the Product Catalog endpoint. Using that fake endpoint, the following code makes the HTTP `GET` request to the Product Catalog microservice.

Listing 2.8 HTTP `GET` request to the Product Catalog microservice

```
language="sql" linenumbering="unnumbered">  public class ProductCatalogClient : IProductCatalogClient
 {
   private readonly HttpClient client;
   private static string productCatalogueBaseUrl =  ❶
     @"https://git.io/JeHiE";
   private static string getProductPathTemplate = "?productIds=[{0}]";

   public ProductCatalogClient(HttpClient client)   ❷
   {
     client.BaseAddress =
         new Uri(productCatalogueBaseUrl);          ❸
     client
       .DefaultRequestHeaders
       .Accept                                      ❹
         .Add(new MediaTypeWithQualityHeaderValue("application/json"));
     this.client = client;
   }

   private async Task<HttpResponseMessage> RequestProductFromProductCatalogue(int[] productCatalogueIds)
   {
     var productsResource =
       string.Format(getProductPathTemplate,
         string.Join(",", productCatalogueIds));
     return await
       this.client.GetAsync(productsResource);      ❺
   }
 }
```

❶ URL of the fake Product Catalog microservice

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

❷ ASP.NET Core injects an HttpClient

❸ Configure the HttpClient to use the base address of the product catalog

❹ Configure the HttpClient to accept JSON responses from the product catalog

❺ Tells HttpClient to perform the HTTP GET asynchronously

This is pretty straightforward. The only thing to note is that by executing the HTTP `GET` request asynchronously, the current thread is freed up to handle other things in Shopping Cart while the request is processed in Product Catalog. This is good practice because it preserves resources in the Shopping Cart microservice, making it a bit less resource intensive and more scalable.

For ASP.NET Core to be able to inject the `HttpClient` in `ProductCatalogClient` we need to register `ProductCatalogClient` as *typed http client* which we do by adding this to `Startup`:

services.AddHttpClient<ProductCatalogClient>();

Now ASP.NET Core can resolve `ProductCatalogClient` which can in turn be injected in other types - the `ShoppingCartController` for instance.

2.2.4 Parsing the product response

The Product Catalog microservice returns product information as a JSON array. The array includes an entry for each requested product, as shown next.

Listing 2.9 Returning a JSON list of products

```
 language="sql" linenumbering="unnumbered">HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

543
[
  {
    "productId": "1",
    "productName": "Basic t-shirt",
    "productDescription": "a quiet t-shirt",
    "price": { "amount" : 40, "currency": "eur" },
    "attributes" : [
    {
      "sizes": [ "s", "m", "l"],
      "colors": ["red", "blue", "green"]
    }]
  },
  {
    "productId": "2",
    "productName": "Fancy shirt",
    "productDescription": "a loud t-shirt",
    "price": { "amount" : 50, "currency": "eur" },
    "attributes" : [
    {
      "sizes": [ "s", "m", "l", "xl"],
      "colors": ["ALL", "Batique"]
    }]
  }
]
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

This JSON must be deserialized, and the information required to create a list of `ShoppingCart` items needs to be read from it. The array returned from Product Catalog is formatted by the microservice's API. To avoid tight coupling between microservices, only the `ProductCatalogClient` class knows anything about the API of the Product Catalog microservice. That means `ProductCatalogClient` is responsible for translating the data received from the microservice into types for the `ShoppingCart` project. In this case, you need a list of `ShoppingCartItem` objects. The following listing shows the code for deserializing and translating the response data.

Listing 2.10 Extracting data from the response

```
language="sql" linenumbering="unnumbered">private static async Task<IEnumerable<ShoppingCartItem>>
     ConvertToShoppingCartItems(HttpResponseMessage response)
   {
     response.EnsureSuccessStatusCode();
     var products =
       JsonConvert.DeserializeObject<List<ProductCatalogueProduct>>(
         await response.Content.ReadAsStringAsync());             ❶
     return
       products
         .Select(p => new ShoppingCartItem(                       ❷
           int.Parse(p.ProductId),
           p.ProductName,
           p.ProductDescription,
           p.Price
       ));
   }

   private class ProductCatalogProduct                            ❸
   {
     public string ProductId { get; set; }
     public string ProductName { get; set; }
     public string ProductDescription { get; set; }
     public Money Price { get; set; }
   }
```

❶ Uses Json.NET to deserialize the JSON from the Product Catalog microservice

❷ Creates a ShoppingCartItem for each product in the response

❸ Uses a private class to represent the product data

If you compare listings 2.9 and 2.10, you may notice that there are more properties in the response than in the `ProductCatalogProduct` class. This is because the Shopping Cart microservice doesn't need all the information, so there's no reason to read the remaining properties. Doing so would only introduce unnecessary coupling. I'll return to this topic in chapters 5, 6, and 8.

The following listing combines the code that requests the product information and the code that parses the response. This method makes the HTTP `GET` request and translates the response to the domain of Shopping Cart.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 2.11 Fetching products and converting them to shopping cart items

```
 language="sql" linenumbering="unnumbered">public async Task<IEnumerable<ShoppingCartItem>>
  GetShoppingCartItems(int[] productCatalogIds)
{
  var response =
    await RequestProductFromProductCatalogue(productCatalogIds);
  return await ConvertToShoppingCartItems(response);
}
```

The `ProductCatalogClient` is almost finished. The only part missing is the code that handles an HTTP request failure.

2.2.5 Adding a failure-handling policy

Remote calls can fail. Not only can they fail, but when running a distributed system at scale, remote calls often do fail. You may not expect the call from Shopping Cart to Product Catalog to fail often, but in an entire system of microservices, there will often be a failing remote call somewhere in the system.

Remote calls fail for many reasons: the network can fail, the call could be malformed, the remote microservice might have a bug, the server where the call is handled may fail during processing, or the remote microservice might be in the middle of a redeploy. In a system of microservices, you must expect failures and design a level of resilience around every place remote calls are made. This is an important topic, and I'll go into more detail in chapter 7.

The level of resilience needed around a particular remote call depends on the business requirements for the microservice making the call. The call to the Product Catalog microservice from the Shopping Cart microservice is important; without the product information, the user can't add items to their shopping cart, which means the e-commerce site can't sell the items to the user. On the other hand, product information doesn't change often, so you could store a copy of it in Shopping Cart and only request it from Product Catalog when the copy doesn't already contain the information. One way to populate such a copy is by caching responses from the Product Catalog. Caching product information has some significant advantages:

- It makes Shopping Cart more resilient to failures in Product Catalog.
- The Shopping Cart microservice will perform better when the product information is present in the cache.
- Fewer calls made from the Shopping Cart microservice mean less stress is put on the Product Catalog microservice.

For now, you won't implement caching; we'll return to the subject of caching for the sake of robustness in chapter 7.

Even with caching in place, some calls from Shopping Cart to Product Catalog are still made.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

For these calls, you may decide that the best strategy for handling failed calls is to retry the call a couple of times and then give up and fail to add any items to the shopping cart. For this chapter, you'll implement a simple retry policy for handling failing requests. You'll use the Polly library, which you'll install in the `ShoppingCart` project as a NuGet Package.

> **NOTE** Polly and failure-handling strategies are described in much more detail in chapter 7.

Using a Polly policy involves these two steps:

1. Declare the policy.
2. Use the policy to execute the remote call.

As you can see in the following listing, Polly's API and integration with ASP.NET Core makes both these steps easy. replace the current registration of `ProductCatalogClient` in `Startup` with this:

Listing 2.12 Microservice error-handling policy

```
language="sql" linenumbering="unnumbered">  services.AddHttpClient<ProductCatalogClient>()
   .AddTransientHttpErrorPolicy(p =>          ❶
     p.WaitAndRetryAsync(                     ❷
       3,
       attempt => TimeSpan.FromMilliseconds(100*Math.Pow(2, attempt))));
```

❶ Wraps calls http calls made in ProductCatalogClient in a Polly policy

❷ Uses Polly's fluent API to set up a retry policy with an exponential back-off

The `HttpClient` injected in `ProductCatalogClient` will use this policy around the call to the Product Catalog microservice: in case of failure, retry the call at most three times. And for each failure, double the amount of waiting time before making the next attempt.

This completes the implementation of `ProductCatalogClient`. Even though in has fewer than 70 lines of code, it does a lot: it builds up the HTTP `GET` request and executes it. It parses the response from Product Catalog and translates it into the shopping cart domain. And it uses the retry policy used for these calls. Next, let's tackle the event feed.

2.2.6 Implementing a basic event feed

The Shopping Cart microservice can now store shopping carts and add items to them. The items include product information from the Product Catalog microservice. Shopping Cart also has an API for other microservices that allows them to add items to or delete items from shopping carts and read the contents of a shopping cart.

The piece missing is the event feed. Shopping Cart needs to publish events about changes to

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

shopping carts, and other microservices can subscribe to these events and react to them as required. In the case of items being added to a shopping cart, figure 2.9 (repeated from chapter 1) illustrates how the Recommendations microservice and the Shopper Tracking microservice base part of their functionality on events from the Shopping Cart microservice.

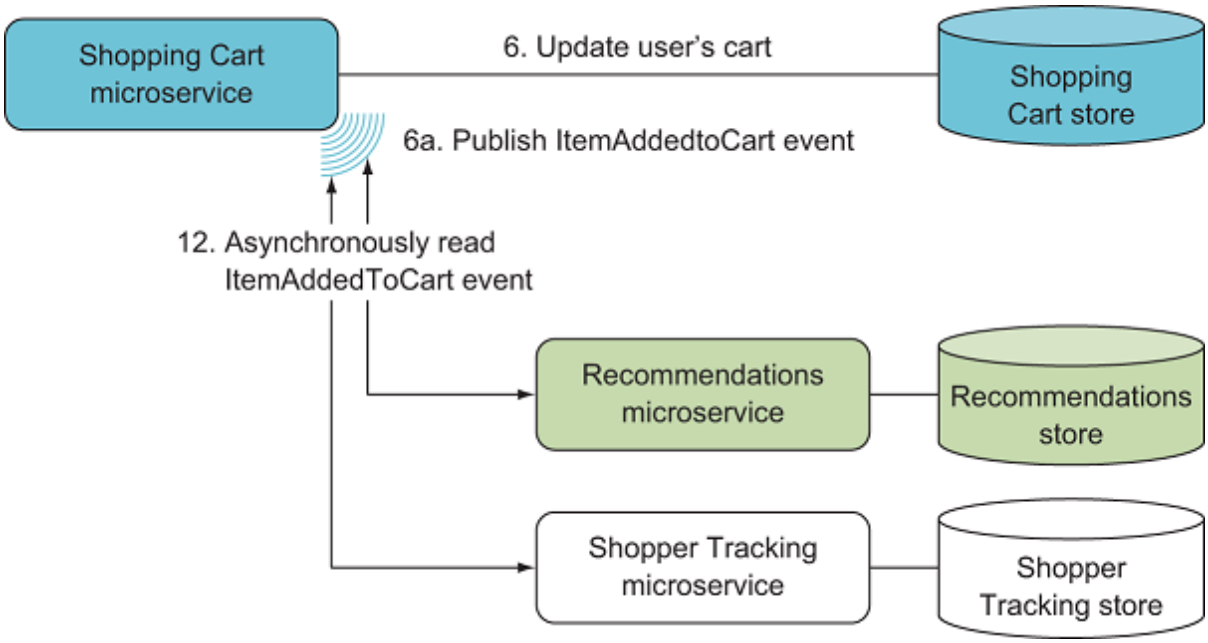

Figure 2.9 The Shopping Cart microservice publishes events about changes to shopping carts to an event feed. The Recommendations and Shopper Tracking microservices subscribe to these events and react as events arrive.

In this section, you'll implement the EventFeed and Shopping Cart domain model components highlighted in figure 2.10. (Chapter 5 returns to the implementation of event feeds and event subscribers.) The domain model is responsible for raising events, and EventFeed allows other microservices to read the events that the Shopping Cart microservice has published.

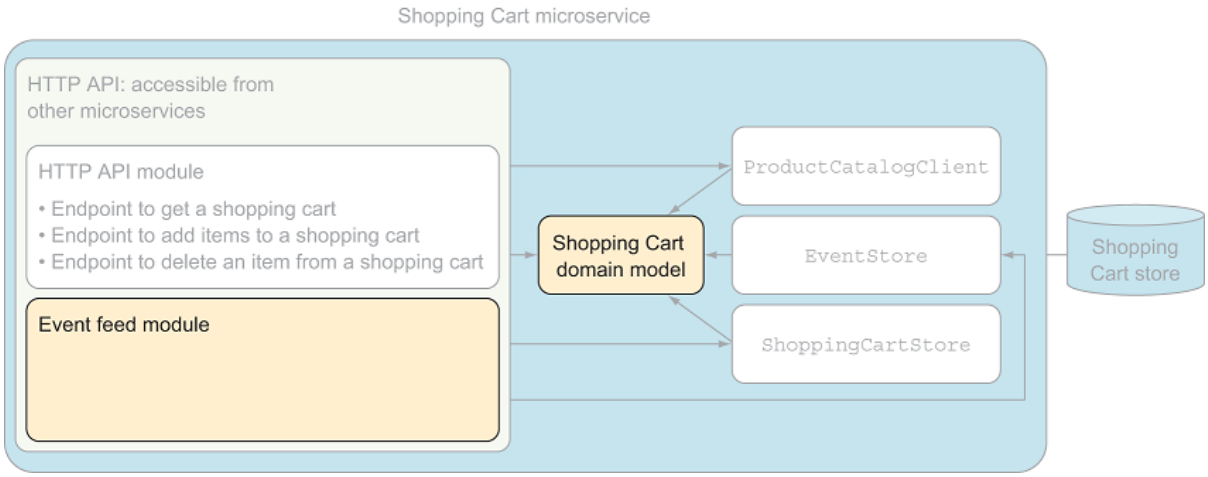

Figure 2.10 The Shopping Cart microservice event feed publishes events to the rest of the e-commerce system.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Implementing the event feed involves these steps:

- *Raise events*. The code in the Shopping Cart domain model raises events when something significant (according to the business rules) happens. Significant events are when items are added to or removed from a shopping cart.
- *Store events*. The events raised by the Shopping Cart domain model are stored in the microservice's data store.
- *Publish events*. Implementing an event feed allows other microservices to subscribe by polling.

We'll work through each of these in turn.

RAISING AN EVENT

In order to be published, events must first be raised. It's usually the domain code in a microservice that raises events, and that's the case in the Shopping Cart microservice. When items are added to a shopping cart, the `ShoppingCart` domain object raises an event by calling the `Raise` method on `IEventStore` and providing the data for the event.

Listing 2.13 Raising events

```
 language="sql" linenumbering="unnumbered">public void AddItems(
      IEnumerable<ShoppingCartItem> shoppingCartItems,
      IEventStore eventStore)
{
  foreach (var item in shoppingCartItems)
    if (this.items.Add(item))
      eventStore.Raise(                          (1)
        "ShoppingCartItemAdded",
        new { UserId, item });
}
```

(1) Raises an event through the eventStore for each item.

From the point of view of the domain code, raising an event is just a matter of calling the `Raise` method on an object that implements the `IEventStore` interface. The `ShoppingCart` domain object also raises an event when an item is deleted. The code for raising that event is almost identical, and I'll leave it to you to implement it.

STORING AN EVENT

The events raised by the domain code aren't published to other microservices directly. Instead, they're stored and then published asynchronously. In other words, all `EventStore` does when an event is raised is store the event in a database, as shown in listing 2.14. As with other database code in this chapter, I'll leave it to your imagination. The important thing to understand is that every event is stored as a separate entry in the event store database, and each event gets a monotonically increasing sequence number.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 2.14 Storing event data in a database

```
 language="sql" linenumbering="unnumbered">public void Raise(string eventName, object content)
{
  var seqNumber = database.NextSequenceNumber();     ❶
  database.Add(
    new Event(
      seqNumber,
      DateTimeOffset.UtcNow,
      eventName,
      content));
}
```

❶ Gets a sequence number for the event

`EventStore` stores every incoming event and keeps track of the order in which they arrive. We'll return to the subject of event stores in chapter 6, where we'll look more at implementing them.

A SIMPLE EVENT FEED

Once events are stored, they're ready to be published—in a sense, they *are* published. Even though one microservice *subscribes* to events from another microservice, an event feed works by having subscribers ask for new events periodically - e.g. once every 30 seconds. Because subscribers are responsible for asking for new events, all you need to do in the Shopping Cart microservice is add an HTTP endpoint that allows subscribers to request events. A subscriber can, for example, issue the following request to get all events newer than event number 100:
GET /events?start=100 HTTP/1.1 Host: shoppingcart.my.company.com Accept: application/json

Or, if the subscriber wants to limit the number of incoming events per call, it can add an `end` argument to the request:

GET /events?start=100&end=200 HTTP/1.1 Host: shoppingcart.my.company.com Accept: application/

Place the implementation of this /events endpoint in a new controller, as shown in the following listing. The endpoint takes an optional starting point and an optional ending point, allowing other microservices to request ranges of events.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 2.15 Exposing events to other microservices

```
 language="sql" linenumbering="unnumbered">namespace ShoppingCart.EventFeed
{
  using System.Linq;
  using Microsoft.AspNetCore.Mvc;

  [Route("/events")]
  public class EventFeedController : Controller
  {
    private readonly IEventStore eventStore;

    public EventFeedController(IEventStore eventStore) => this.eventStore = eventStore;

    [HttpGet("")]
    public Event[] Get(
      [FromQuery] long start,                    ❶
      [FromQuery] long end = long.MaxValue)
    =>
      this.eventStore
        .GetEvents(start, end)                   ❷
        .ToArray();
  }
}
```

❶ Reads the start and end values from a query string parameter

❷ Returns the raw list of events. MVC takes care of serializing the events into the response body.

`EventFeedController` mostly uses MVC features that you've already encountered. The only new bit is that the `start` and `end` values are read from query string parameters. The `FromQuery` parameter in front of the method arguments tells MVC that these are query string parameters.

`EventFeedController` uses the event store to filter out events between the `start` and `end` values from the client. Although filtering is probably best done at the database level, the following simple implementation illustrates it well.

Listing 2.16 Filtering events based on the start and end points

```
 language="sql" linenumbering="unnumbered">public IEnumerable<Event> GetEvents(
  long firstEventSequenceNumber,
  long lastEventSequenceNumber)  =>
    database
      .Where(e =>
        e.SequenceNumber >= firstEventSequenceNumber &&
        e.SequenceNumber <= lastEventSequenceNumber)
      .OrderBy(e => e.SequenceNumber);
```

With the /events endpoint in place, microservices that want to subscribe to events from the Shopping Cart microservice can do so by polling the endpoint. Subscribers can—and should—use the `start` and `end` query string parameters to make sure they only get new events. If Shopping Cart is down when a subscriber polls, the subscriber can ask for the same events again later. Likewise, if a subscriber goes down for a while, it can catch up with events from

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Shopping Cart by asking for events starting from the last event it saw. As mentioned, this isn't a full-fledged implementation of an event feed, but it gets you to the point that microservices can subscribe to events, and the code is simple.

You've now completed the version 1 implementation of your first microservice. As you can see, a microservice is small and has a narrow focus: it provides just one business capability. You can also see that microservice code tends to be simple and easy to understand. This is why you can expect to create new microservices and replace existing ones quickly.

2.3 Running the code

Now that all the code for the Shopping Cart microservice is in place, you can run it the same way you ran the example in chapter 1: from within Visual Studio, or from the command line with `dotnet`. You can test out all the endpoints with RestClient or a similar tool. When you first try to fetch a shopping cart with an HTTP `GET` to /shoppingcart/123, the cart will be empty. Try adding some items to it with an HTTP `POST` to /shoppingcart/123/items and then fetching it again; the response should contain the added items. You can also look at the event feed at /events, and you should see events for each added item.

> **WARNING** I haven't shown implementations of `EventStore` or `ShoppingCartStore`. If you haven't created your own implementations of these, your microservice won't work.

2.4 Summary

- Implementing a complete microservice doesn't take much code. The Shopping Cart microservice has only the following:
  - Two short controllers
  - A simple `ShoppingCart` domain class
  - A client class for calling the Product Catalog microservice
  - Two straightforward data access classes: `ShoppingCartDataStore` and `EventStore` (not shown in this chapter)
- MVC makes it simple to implement HTTP APIs. The routing attributes MVC provides makes it easy to add endpoints to a microservice. Just add an action methods and put a routing attribute - like `[HttpGet("")]` or `[HttpPost("")]` on it.
- You should always expect that other microservices may be down. To prevent errors from propagating, each remote call should be wrapped in a policy for handling failure.
- The Polly library is useful for implementing failure-handling policies and wrapping them around remote calls.
- Implementing a basic event feed is simple and enables other microservices to react to events. The poor man's event feed implemented in this chapter is just a short controller.
- Domain model code is usually responsible for raising events, which are then stored in an event store and published through an event feed.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

3 Deploying a microservice to Kubernetes

This chapter covers

- Packaging a microservices in a Docker container
- Deploying a microservice container to Kubernetes on localhost
- Creating a basic Kubernetes cluster on Azure's AKS (Azure Kubernetes Service)
- Deploying a microservice container to a Kubernetes cluster on AKS

In chapter 2 we developed a simple shopping cart microservice, but we only ran it on localhost directly with the `dotnet run` command. In this chapter we take that microservice and deploy it to a production-like environment in a public cloud. Our microservices can run in many different environments, but we are going to pick just one as an example. We are going to focus on how to take the shopping cart microservice and run it in a Kubernetes cluster in Microsoft Azure. To reach that goal in this chapter we will:

- Put the shopping cart into a Docker container
- Setup Kubernetes on localhost, so we have a testing ground
- Setup Kubernetes in Azure, so we have a production-like environment
- Run the same shopping cart container in the localhost Kubernetes cluster and the Azure Kubernetes cluster

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Docker containers**

Containers are a way of wrapping an application - or in our case a microservice - in a portable image that brings along everything it needs to run from operating system to application code to library dependencies. A container image can be built once, moved to different environments and still be expected to work - as long as the application code in the container does not make assumption about the environment. In this sense containers are similar to virtual machines. But where virtual machines virtualize the hardware containers virtualize the operating systems, which allows containers to be much smaller. This is important in a microservice context where we are going to have many, many microservices. With containers we are able to run many microservices on the same server while maintaining a good level of isolation between them.

We will get to the implementation of these steps soon, but first we will discuss why this is a solid approach and touch upon what some of the alternatives are.

3.1 Choosing a production environment

Simply running our microservices on localhost isn't very interesting. We need them to run somewhere our end users can get to them and use them. There are a number of options for doing that including:

- Running the microservices on your own Windows or Linux servers on-premises. The microservices we write are .NET Core application which mean they readily run on both Windows and Linux. The HTTP APIs we create are ASP.NET Core apps and use ASP.NET Core's web server, Kestrel, which can be put behind a proxy like Nginx or IIS. So if you or organization prefers an on-premises solution the microservices we create can be hosted in a fairly traditional on-premises environment.
- Using a Platform as a Service (PaaS) cloud option that support .NET Core like Azure Web Apps or Azure Service Fabric. Using a PaaS option means that you are no longer maintaining the underlying infrastructure yourself. You only have to maintain your microservices. This does not mean that you have no operations work, but the operations that you have to do is focused on your microservices, not the underlying infrastructure. E.g. you do not need to keep an operating system patched or worry about renewing hard drives as they age. You only have to deal with what the code in the microservices does.
- Putting microservices into containers and deploying them to a cloud specific container service like Azure's ACS or Amazon's ECS. As we will see in this chapter our microservices can easily be put into containers. Both ACS and ECS can run these containers and offer tooling to manage the containers. Like a PaaS option this frees you from maintaining the underlying infrastructure, but since containers come with an OS you will have to keep that up-to-date and patched.
- Using cloud agnostic container orchestrators like Kubernetes, Apache Mesos or RedHat OpenShift. Again containerizing our microservices is easy and once they are in containers any container orchestrator can run them. There is wide variability in this space

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

and lumping Kubernetes, Mesos and OpenShift together might not be completely fair, but they can all run and manage our microservices and they are not tied to any particular cloud provider.

Throughout the book we take care to keep these options open. On the other hand I also want to show you how to get the microservices running in a production-like environment. That means choosing one of the options listed above to use as the example of a production environment in this chapter and in the remainder of the book. This does not mean that the other options are not viable; they are.

For the purpose of this book I choose to use containers and to run them in Kubernetes because it gives us a number of benefits that dovetail nicely with the flexibility and scalability we are aiming for with microservices:

- By choosing to put our microservices in containers we keep several of the options listed above open, which gives us flexibility.
- By choosing Kubernetes we get a mature and widely used container orchestrator which is supported by all the major clouds and which can also run on your own servers. Kubernetes has a mature and large eco-system, is highly scalable and built for systems that consist of many small containers that are often updated and re-deployed, just like the microservices system we are talking about.

NOTE Make sure you've set up Docker on your development environment. In appendix A, you'll find instruction to install Docker.

The setup we are going to build is illustrated in figure 3.1. We create a Kubernetes cluster in AKS with a single node. In that cluster we will deploy a load balancer and an instance of the Shopping Cart container. The load balancer will have a public endpoint that will take traffic from the outside, and will route that traffic to the Shopping Cart container.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

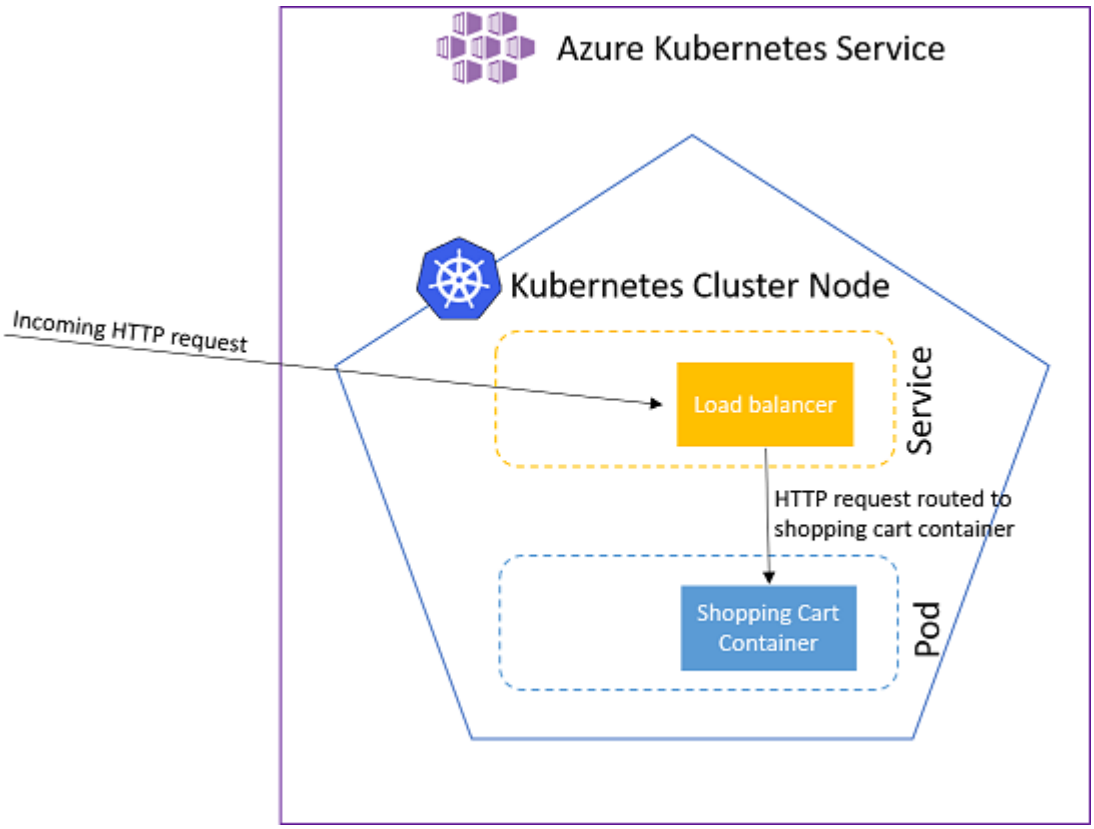

Figure 3.1 We will create a single node Kubernetes cluster in AKS and deploy a load balancer and a Shopping Cart container to it.

Now that we have settled on using Kubernetes let's get Shopping Cart running in Kubernetes - first on localhost, then on Azure.

3.2 Putting the shopping cart microservice in a container

The first step towards running the Shopping Cart microservice in Kubernetes is to put it into a container and run that container. We continue using the Shopping Cart microservice that we developed in chapter 2.

3.2.1 Add a Dockerfile to the Shopping Cart microservice

To put Shopping Cart microservice in a container we first need to add a *Dockerfile* to the root folder of the Shopping Cart project next to the solution file. The Dockerfile is a description of the container we want to build and should look like this:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 3.1 Dockerfile for the Shopping Cart

```
 language="sql" linenumbering="unnumbered">FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
❶
WORKDIR /src
COPY ["ShoppingCart/ShoppingCart.csproj", "ShoppingCart/"]
RUN dotnet restore "ShoppingCart/ShoppingCart.csproj" ❷
COPY . .
WORKDIR "/src/ShoppingCart"
RUN dotnet build "ShoppingCart.csproj" -c Release -o /app/build ❸

FROM build AS publish
RUN dotnet publish "ShoppingCart.csproj" -c Release -o /app/publish ❹

FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS final  ❺
WORKDIR /app
EXPOSE 80                                               ❻
COPY --from=publish /app/publish .                      ❼
ENTRYPOINT ["dotnet", "ShoppingCart.dll"]               ❽
```

❶ The .NET Core SDK used for building the microservice

❷ Restore NuGet packages

❸ Build the microservice in release mode

❹ Publish the microservice to the /app/publish folder

❺ The image the final container is based on

❻ The container should accept requests on port 80

❼ Copy files from /app/publish to final container

❽ Specify that when the final container runs it will start up `dotnet ShoppingCart.dll`

There is a lot going on in that Dockerfile, so lets unpack it piece by piece. The first thing to understand is that the Dockerfile describes a *multi-stage build* which means that there are multiple discrete steps in the file. The end result is a ready to run container image with the compiled Shopping Cart microservice. The steps in the Dockerfile are:

- Building the Shopping Cart code: The first part of the Dockerfile builds the Shopping Cart code using a docker image that contains the .NET Core SDK and calling first `dotnet restore` and then `dotnet build`. This is just like we have already done in chapter 2 to build the Shopping Cart locally except we use a couple of extra options to indicate that we want a Release build and to specify output folders.
- Publish the Shopping Cart microservice: The second part of the Dockerfile uses the `dotnet publish` command to copy the files needed at runtime from the build output folder ("/app/build") to a new folder called "/app/publish".
- Create a container image based on ASP.NET Core: The third and final step in the Dockerfile creates the final container image which is the result of the multi-stage build described in the Dockerfile. The step is based on an ASP.NET Core docker image from

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Microsoft. That image comes with the ASP.NET Core runtime. We add the files from the "/app/publish" folder and specify that the entry point to the container image is `dotnet ShoppingCart.dll` which is a command that runs the compiled ASP.NET Core application in the ShoppingCart.dll.

To make sure the Dockerfile runs a clean build we add a `.dockerignore` with these lines that make sure any `bin` and `obj` folders are not copied into the container:

[B|b]in/ [O|o]bj/

With this Dockerfile in place we are ready to build a Shopping Cart container image and to run it.

3.2.2 Build and run the shopping cart container

The next step is to build a Shopping Cart container image from the Dockerfile we just added. To do so open a command line and go to the root of Shopping Cart - where the Dockerfile is. Then issue this Docker command:
> docker build . -t shopping-cart

This can take a while the fist time. Be patient. Subsequent builds will be faster. The output of the `docker build` is rather long. When it is successful the last few lines of the output are similar to:

Successfully built 8d448ba53088 Successfully tagged shopping-cart:latest

Possibly followed by this warning if you are on Windows:

SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Do
files and directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to dou
and reset permissions for sensitive files and directories.

This warning comes becomes because you are working with Linux containers on Windows, which is fine for a development environment. The production-like environment we will set up on Azure will be based on Linux servers, so we can safely ignore the warning.

You are now ready to run the newly build container image with this command:

> docker run --rm -p 5000:80 shopping-cart

This starts the Shopping Cart container and maps port 80 inside the container to port 5000 outside the container as shown in figure 3.2. This means that you can access the Shopping Cart on `http://localhost:5000` and use the endpoints the same way we did in chapter 2.

The details of the command above are that:

- `--rm` means that the container is automatically removed when the container exists. This is nice during development to avoid cluttering your localhost machine with too many old containers

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

- `-p 5000:80` means that the container exposes port 5000 and listens to traffic on that port. Any incoming traffic to port 5000 is forwarded to port 80 inside the container
- 'shopping-cart' is the name of the container image to run

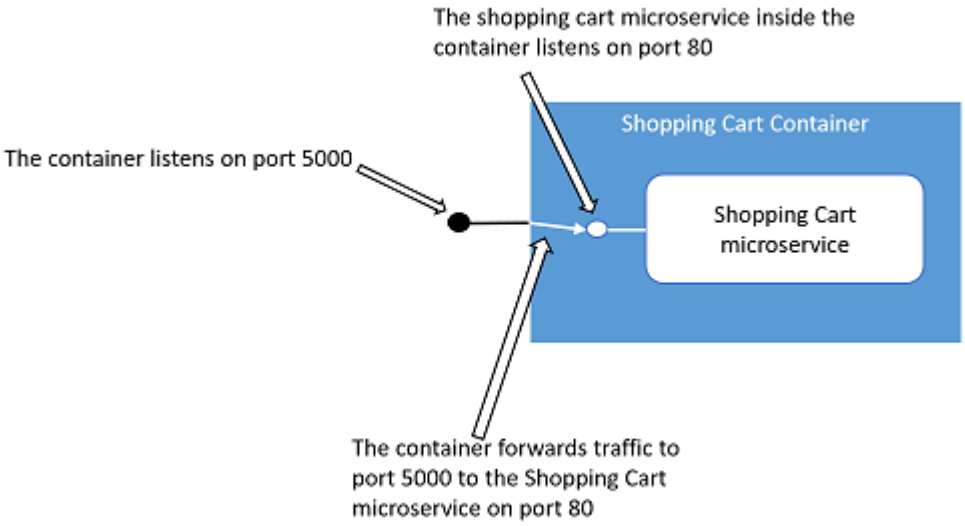

Figure 3.2 The shopping cart listens to port 80 inside the container. The container listen to traffic on port 5000 and forwards that traffic to the Shopping Cart microservice.

This confirms that we have successfully build a container with the Shopping Cart. All we had to do was create a Dockerfile that describes how to build the container and then use Docker to build it. From that we got a portable container image. This is quite nice because it enables us to run the exact same container image in all the environments we want including both localhost and production. Since we can run the same image across different environments we can be reasonably confident that it will work the same across environments.

Looking at the Dockerfile we can notice that there nothing about it that makes it specific to the Shopping Cart, expect the names of the `.csproj` file and the `.dll` file. So not only have we built a Shopping Cart container that we can run in a variety of environments we have learned how to do the same for all the other .NET Core based microservices we will build.

Next thing we need is to run the Shopping Cart container in Kubernetes.

3.3 Running the shopping cart container in Kubernetes

Now that the Shopping Cart is in a container it is ready to run in Kubernetes. We will create a Kubernetes manifest file and use it to run the Shopping on Kubernetes - first localhost and then on Azure. Running the Shopping Cart container in Kubernetes looks like figure 3.3 which is almost exactly like the diagram at the beginning of the chapter except Kubernetes is running on localhost instead of on Azure. This is no accident, since we are going run the Shopping in the same way localhost as we will on Azure.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

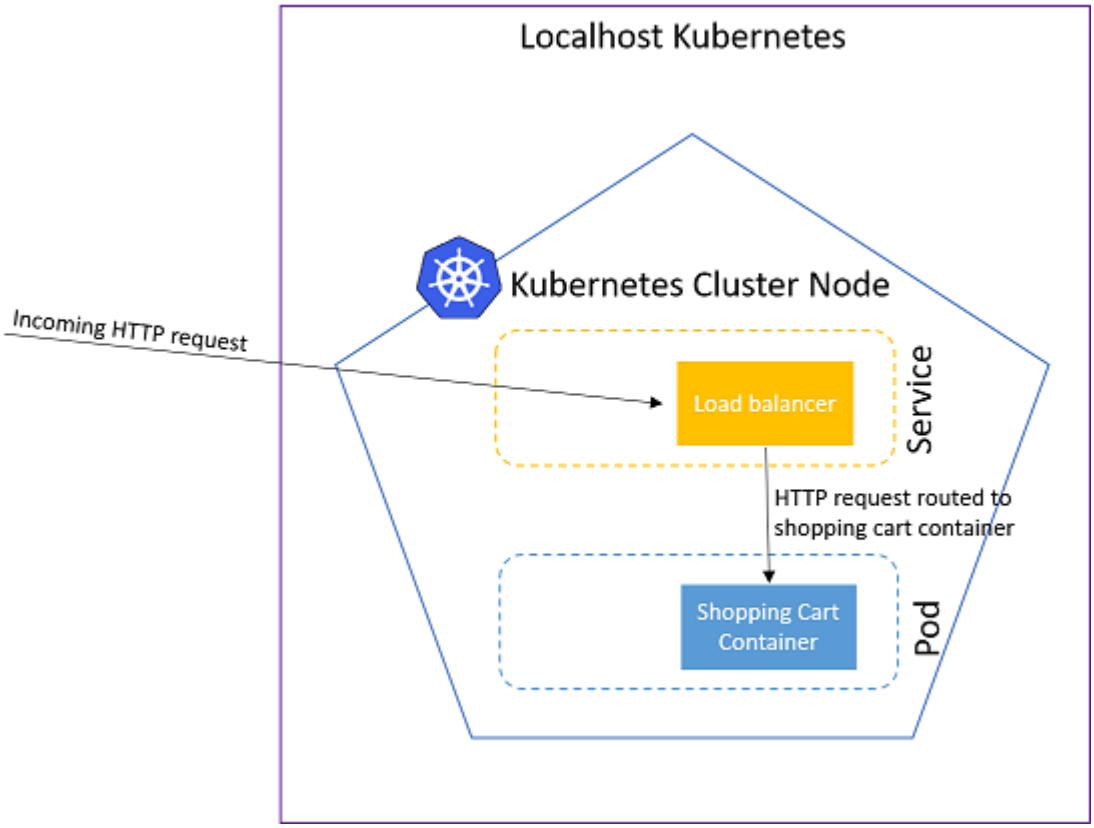

Figure 3.3 Running the Shopping Cart on Kubernetes on localhost is almost the same as on Azure

3.3.1 Set up Kubernetes localhost

How to set up Kubernetes on your development machine depends the operating system you are using. If you are on Windows or Mac I recommend using Docker Desktop which comes with the option to enable Kubernetes. If you are on Linux there are numerous options; one easy option is MicroK8S.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Enabling Kubernetes in Docker Desktop**

To enable Kubernetes in Docker Desktop open the Docker Desktop settings, choose Kubernetes and click "Enable Kubernetes", as show in figure 3.4. This will take a while the first time, since Docker Desktop will download, install, and start Kubernetes.

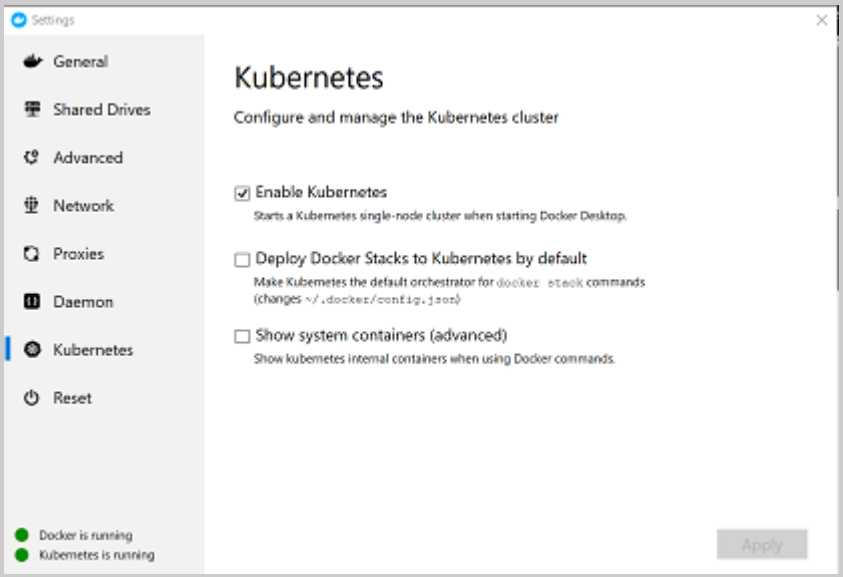

Figure 3.4 Docker Desktop Kubernetes settings

Once the UI indicates that Kubernetes is running you are ready to start working with Kubernetes on your localhost.

SIDEBAR **Installing and starting MicroK8S**

To install MicroK8S on a Linux machine simply run this command:

> sudo snap install microk8s --classic

This will install and start the Kubernetes cluster. Furthermore this installs the `microk8s` command line interface which includes the `kubectl` command that we are going to use. To make it a bit easier to follow along with the work in the following sections you can create an alias for kubectl:

> snap alias microk8s.kubectl kubectl

When Kubernetes is running you can go to the command line and check that Kubernetes is indeed running, using this command

> kubectl cluster-info

Which should give you a response similar to this:

Kubernetes master is running at https://kubernetes.docker.internal:6443 KubeDNS is running at https://kubernetes.docker.internal:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy T

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

The `kubectl` command is the command line interface to control Kubernetes and we will be using that to deploy the Shopping Cart to Kubernetes both localhost and on Azure. Furthermore `kubectl` can be used to inspect the Kubernetes cluster and to start the Kubernetes Dashboard which gives you a friendly UI for looking inside the Kubernetes cluster.

3.3.2 Create Kubernetes deployment for the shopping cart

Next we want to deploy the Shopping Cart to the Kubernetes cluster we just installed and started. To do that we need add a manifest file describing the deployment to the Shopping Cart code base called `shopping-cart.yaml`. The `shopping-cart.yaml` file contains two major section:

1. A deployment section that specifies which container we want to deploy and how we want it setup:
   - We want to deploy the Shopping Cart container
   - We want 1 copy of the container running. We could set this number differently and Kubernetes would take care of running as many instances as we wanted.
   - The port the container communicates on - port 80 in the case of the Shopping Cart just like we saw earlier
2. A service section that configures load balancing in front of the Shopping Cart. The load balancer makes the Shopping Cart accessible outside the Kubernetes cluster by giving it an external IP. If we were deploying more than one instance of the Shopping Cart container the load balancer would balance the incoming traffic between the Shopping Cart instances.

This is contents of the `shopping-cart.yaml` file:

kind: Deployment apiVersion: apps/v1 metadata: name: shopping-cart spec: replicas: 1 selector: matchl shopping-cart template: metadata: labels: app: shopping-cart spec: containers: - name: shopping-cart im shopping-cart imagePullPolicy: IfNotPresent ports: - containerPort: 80 --- kind: Service apiVersion: v1 name: shopping-cart spec: type: LoadBalancer ports: - name: shopping-cart port: 5000 targetPort: 80 se shopping-cart

Using this manifest to deploy and run the Shopping Cart in Kubernetes is as simple as running this from the command line:

> kubectl apply -f shopping-cart.yaml

This command tells Kubernetes to run everything described in the manifest, so Kubernetes will start the load balancer and the Shopping Cart. To check if the both these are running we can use the command `kubectl get all` which will list everything running in the Kubernetes cluster. If the deployment went well the output from `kubectl get all` should be similar to this:

NAME READY STATUS RESTARTS AGE pod/shopping-cart-f4c8f4b94-4j48v 1/1 Running 0 15h N CLUSTER-IP EXTERNAL-IP PORT(S) AGE service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCI

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

service/shopping-cart LoadBalancer 10.103.8.64 localhost 5000:31593/TCP 6d20h NAME READY UI
AVAILABLE AGE deployment.apps/shopping-cart 1/1 1 1 6d20h NAME DESIRED CURRENT REA
replicaset.apps/shopping-cart-f4c8f4b94 1 1 1 15h

If you prefer to have a UI you can install and start the Kubernetes Dashboard. First install the Kubernetes dashboard using this command:

> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta8/aio/deploy/recommended.yaml

Now running the command `kubectl proxy` will start the dashboard and make it available on the somewhat gnarly URL `http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kuber`. This dashboard should also show that the Shopping Cart is running as shown in figure 3.5.

> **NOTE** If the Kubernetes dashboard asks for a login token you can get one by running command `kubectl -n kube-system describe secret default` and copying the token from the response.

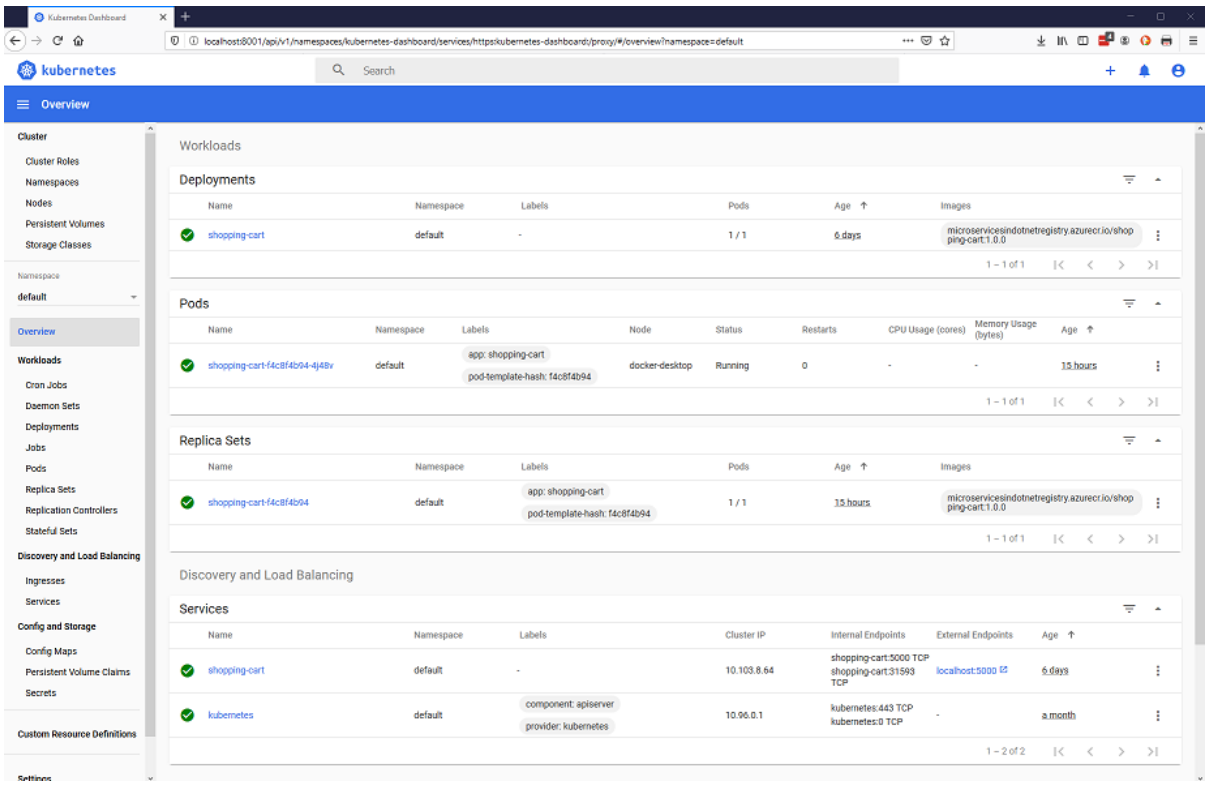

Figure 3.5 Docker Desktop Kubernetes settings

We are pretty sure Kubernetes is running the Shopping Cart just fine based on what we see in the dashboard and from `kubectl`, but to make sure it runs as expected lets test the Shopping Cart's endpoints. First let's add a couple of item to a cart:

POST http://localhost:5000///shoppingcart/15/items Accept: application/json Content-Type: applicatior

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Next let's read the same cart back:

GET http://localhost:5000///shoppingcart/15

The body of the response from the GET request should be the list of items in the cart:

{ "userId": 15, "items": [{ "productCatalogueId": 1, "productName": "Basic t-shirt", "description": "a c
"price": { "currency": "eur", "amount": 40 } }, { "productCatalogueId": 2, "productName": "Fancy shir
"description": "a loud t-shirt", "price": { "currency": "eur", "amount": 50 } }] }

This tells us that the Shopping Cart indeed works on Kubernetes too just as we expected since it is the exact same container image that we already ran and tested earlier.

Looking at the Kubernetes manifest we used to run the Shopping Cart on Kubernetes we can notice that apart from the name `ShoppingCart` the manifest would also work for other microservices. This means that we now have a template for creating Dockerfiles for out microservices and for creating Kubernetes manifests for them. This puts us in a very good position to quickly get all our microservices deployed to Kubernetes too.

We are still on localhost, though, so let's move ahead get the Shopping Cart up and running in Azure.

3.4 Running the shopping cart container in Azure Kubernetes Service

The next goal is to run the Shopping Cart on Azure Kubernetes Service (AKS). When we have done that at the end of this section we will have reached the goal of running our first microservice in a production-like environment and will have created the setup illustrated in figure 3.1. In order to get there we need to:

- Set up AKS: We need to create all the Azure resources for a Kubernetes cluster in AKS. This includes the cluster itself, networking and a private container registry where we will store the container images for our microservices
- Pushing the Shopping Cart container image to our private container registry
- Deploying the Shopping Cart to the AKS cluster using the Shopping Cart's Kubernetes manifest

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Alternatives to Azure Kubernetes Service**

Azure is by no means the only place we can run Kubernetes. The other major public clouds - Google Cloud, Amazon Web Services, Digital Ocean and others - offer similar managed Kubernetes services. All of these are easy to set up and they all offer tools to scale and monitor the clusters. The reason I use Azure is familiarity, the others are equally viable options.

If you prefer to manage your own servers you can also choose to set up Kubernetes on your own cloud or on-premises servers. The differences compared to the managed Kubernetes services are in the setup and management of the Kubernetes cluster. The management of the containers we deploy to Kubernetes is the same in all cases.

3.4.1 Set up AKS

The setup we need in order to be ready to work with a Kubernetes cluster in AKS consists of four parts:

1. Creating a *resource group* in Azure. A resource group is just a grouping of things in Azure. We are putting all the resources for the Kubernetes cluster and the container registry in one resource group. This will group everything together in the Azure portal and will make it easy for us to clean everything up once we are done.
2. Creating a private container registry
3. Creating a Kubernetes cluster in AKS. We will create a small cluster with just one node, and we attach it to the private container registry to allow Kubernetes to pull containers from our registry.
4. Log our local Kubernetes command line - `kubectl` - into the newly created AKS cluster

Each of these four parts are performed by an Azure command with the Azure command line tool `az`. The end result in Azure is as shown in figure 3.6 an Azure resource group with a private Azure Container Registry and single node Kubernetes cluster in AKS. This is our production environment.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

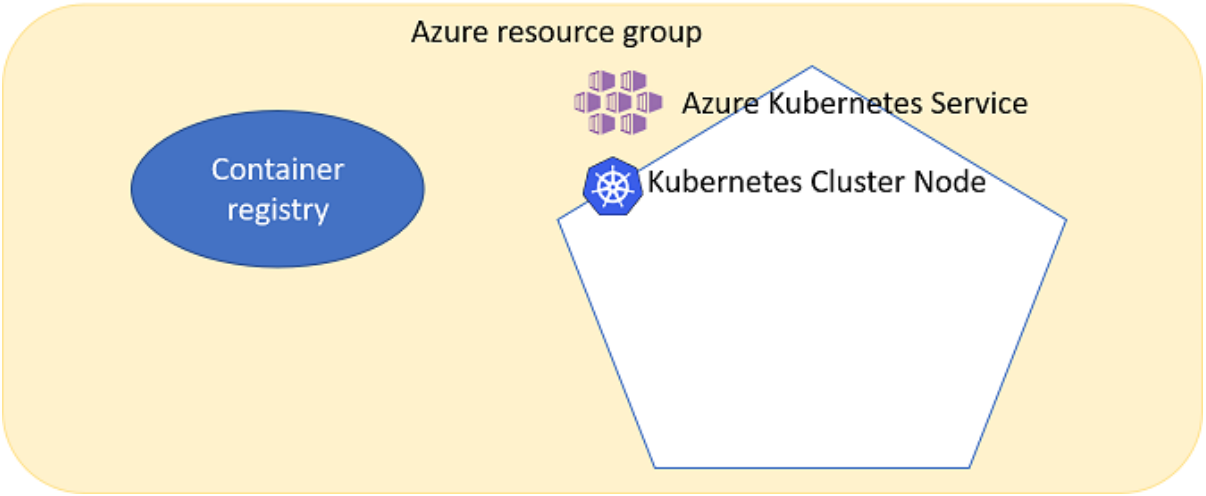

Figure 3.6 Running the Shopping Cart on Kubernetes on localhost is almost the same as on Azure

> **NOTE** To follow along with the set up of AKS you need to have the Azure CLI installed. In appendix A you will find instructions on installing the Azure CLI.

To make the AKS setup easy to repeat I have gathered the commands needed to setup AKS in a PowerShell script called `create-aks.ps1` that looks like this:

```
az group create --name MicroservicesInDotnet --location northeurope az acr create --resource-group
MicroservicesInDotnet --name MicroservicesInDotnetRegistry --sku Basic az aks create --resource-gro
MicroservicesInDotnet --name MicroservicesInDotnetAKSCluster --node-count 1 --enable-addons mor
--generate-ssh-keys --attach-acr MicroservicesInDotnetRegistry az aks get-credentials --resource-group
MicroservicesInDotnet --name MicroservicesInDotnetAKSCluster kubectl get nodes
```

The first four lines correspond exactly to the four parts listed above: creating a resource group, creating a container registry, creating the AKS cluster, and logging `kubectl` into the cluster. The last line shows the nodes in the cluster, so we know the cluster is up and running.

> **NOTE** The `az acr create …` command needs a unique registry name. If the name is not unique you get an error about the name already being in use. You can check if the name is already claimed using API documented here: https://docs.microsoft.com/en-us/rest/api/containerregistry/registries/checknameavailab

Running the script takes a while - about 5 - 10 minutes. It also produces a lot of output although in burst, so don't be surprised if there no feedback for a few minutes followed by a burst of output. The last lines of the output is the output from the line `kubectl get nodes` and should look similar to this:

```
NAME STATUS ROLES AGE VERSION aks-nodepool1-32786309-vmss000000 Ready agent 107s v
```

This a list of the nodes in the Kubernetes cluster your `kubectl` command line is attached to. The

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

name of your node will differ from mine, but should start with `aks-`.

> **NOTE** If you want some more background and detail on the steps involved in setting up AKS I refer to Microsofts documentation, for instance the article "Quickstart: Deploy an Azure Kubernetes Service cluster using the Azure CLI" at https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough

When at some point you are done with the Kubernetes cluster in AKS you can delete the cluster as well as the container registry with this command:

```
> az group delete --name MicroservicesInDotnet --yes --no-wait
```

At this point the Kubernetes cluster in AKS is up and running. Before we move on, we will get the Kubernetes Dashboard up and running too. There are two steps towards running the Kubernetes Dashboard: First we must allow the dashboard to access the cluster and then we start the dashboard.

This command will give the dashboard access, by telling Kubernetes to assign the role `cluster-admin` to the account `kube-system:kubernetes-dashboard`:

```
> kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin
--serviceaccount=kube-system:kubernetes-dashboard
```

The Kubernetes dashboard uses the `kube-system:kubernetes-dashboard` account, so now it will have cluster admin rights.

With the access in place we can start the Kubernetes dashboard with the Azure command line like this:

```
> az aks browse --resource-group MicroservicesInDotnet --name MicroservicesInDotnetAKSCluster
```

This will start the Kubernetes Dashboard and open a browser tab with the dashboard as shown in figure 3.7

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

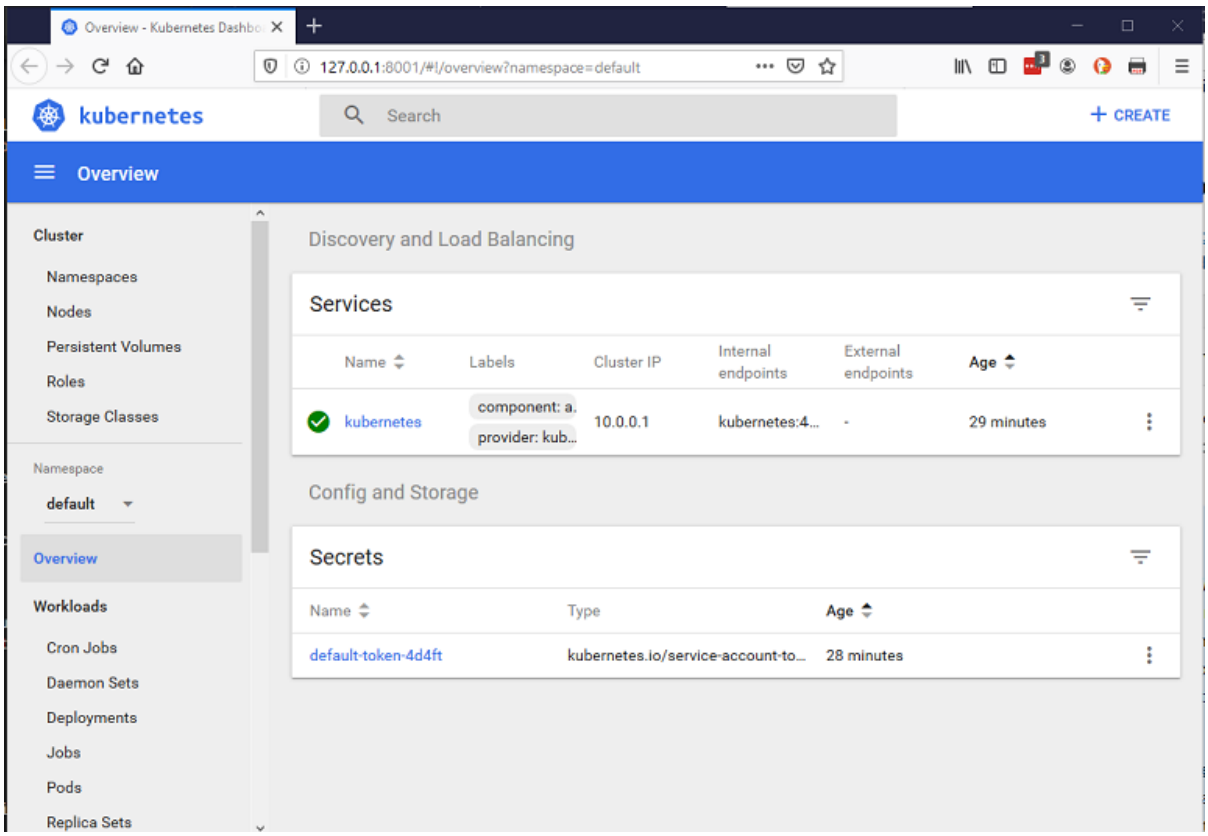

Figure 3.7 The Kubernetes Dashboard

NOTE If you have any trouble with getting the Kubernetes Dashboard up and running I refer you to Microsofts documentation which has the article "Access the Kubernetes web dashboard in Azure Kubernetes Service (AKS)" at https://docs.microsoft.com/en-us/azure/aks/kubernetes-dashboard which has troubleshooting information.

Now we have a Kubernetes cluster running in Azure and we have the dashboard running, so we have a friendly interface to what's running the cluster. We also have the Shopping Cart container image and a Kubernetes manifest file for the Shopping Cart. With all this in place we are ready to deploy the Shopping Cart to our AKS cluster. Not only that. Since the way we built the Shopping Cart container easily extends to building container for other microservices and since the Kubernetes manifest also easily extends to other microservices we are ready to deploy the microservice we will build in the upcoming chapter to AKS too.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

3.4.2 Run the shopping cart in AKS

With the infrastructure to run microservices in AKS in place we return to the Shopping Cart microservice. We have already put the Shopping Cart into a container and we have run that container in Kubernetes on localhost by applying the Kubernetes manifest for the Shopping Cart to the localhost Kubernetes. Deploying the Shopping Cart to AKS is mostly re-using the things we have already made, but there a few tweaks. The remaining steps to deploying and running the Shopping Cart in AKS are:

- Tag the Shopping Cart container image, so we have a precise identification of the container image
- Push the tagged container image to our container registry in Azure
- Modify the Shopping Cart's Kubernetes manifest to refer to the tagged container
- Apply the modified manifest to AKS
- Test that the Shopping Cart runs correctly

As listed above we need to tag the Shopping Cart container image that we built in section "Build and run the shopping cart container". The tag we will give the container is `microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0`. The first part of the tag is the address of our container registry in Azure - `microservicesindotnetregistry.azurecr.io` - the second part is the name and version of the container image. We set the tag as follows:

```
> docker tag shopping-cart microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0
```

We have to push the container image to a container registry to make it available for AKS to pull down and run. AKS cannot pull a container image from my or your localhost. The registry we use in Azure is private, so any service pulling from it or pushing to it must be authenticated. When we created the AKS cluster we used the option `--attach-acr MicroservicesInDotnetRegistry` which connects the AKS cluster with the container registry called MicroservicesInDotnetRegistry and allows AKS to pull images. Likewise the docker client on our localhost needs to be authenticated with the container registry before it is allowed to push container images to the registry. This authentication is done using the Azure cli like this:

```
> az acr login --name MicroservicesInDotnetRegistry
```

Now the tagged container image can be pushed to our private docker registry in Azure with this Docker command:

```
> docker push microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0
```

The Shopping Cart container image is now available for our AKS cluster to pull, but we have to tell it to do so. To that end we have to modify the manifest file, `shopping-cart.yaml`, to refer to the container by the tag

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

`microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0`, which means the manifest becomes:

kind: Deployment apiVersion: apps/v1 metadata: name: shopping-cart spec: replicas: 1 selector: matchl shopping-cart template: metadata: labels: app: shopping-cart spec: containers: - name: shopping-cart im microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0 imagePullPolicy: IfNotPresent ports: - cor 80 --- apiVersion: v1 kind: Service metadata: name: shopping-cart spec: type: LoadBalancer ports: - na shopping-cart port: 5000 targetPort: 80 selector: app: shopping-cart

The only change is the image name which now refers to the image in our private container registry in Azure. Now all that remains is to apply this manifest to the Kubernetes cluster in AKS:

> kubectl apply -f shopping-cart.yaml

After a short while the shopping cart will be running in AKS. In order verify this we need the IP address of the Shopping Cart in AKS, so we can call it's endpoints. We can either find the IP address of the Shopping Cart in the Kubernetes dashboard or with the command line.

Using the command line the command `kubectl get all` will show the information about everything running in the cluster. The output should be similar to this:

NAME READY STATUS RESTARTS AGE pod/shopping-cart-f4c8f4b94-vnmn9 1/1 Running 0 107s TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE service/kubernetes ClusterIP 10.0.0.1 <none> 44 service/shopping-cart LoadBalancer 10.0.100.183 52.142.83.184 5000:31552/TCP 107s NAME READ UP-TO-DATE AVAILABLE AGE deployment.apps/shopping-cart 1/1 1 1 107s NAME DESIRED CU READY AGE replicaset.apps/shopping-cart-f4c8f4b94 1 1 1 107s

The IP address of the shopping cart load balancer is in the external IP column in the list of services. In this case the IP address is `52.142.83.184` and the port is `5000` just like it was on localhost. We can call the Shopping Cart through the load balancer, in this case at `52.142.83.184:5000`.

Similarly the IP address can be found in the Kubernetes Dashboard under "Services" as shown in figure 3.8

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

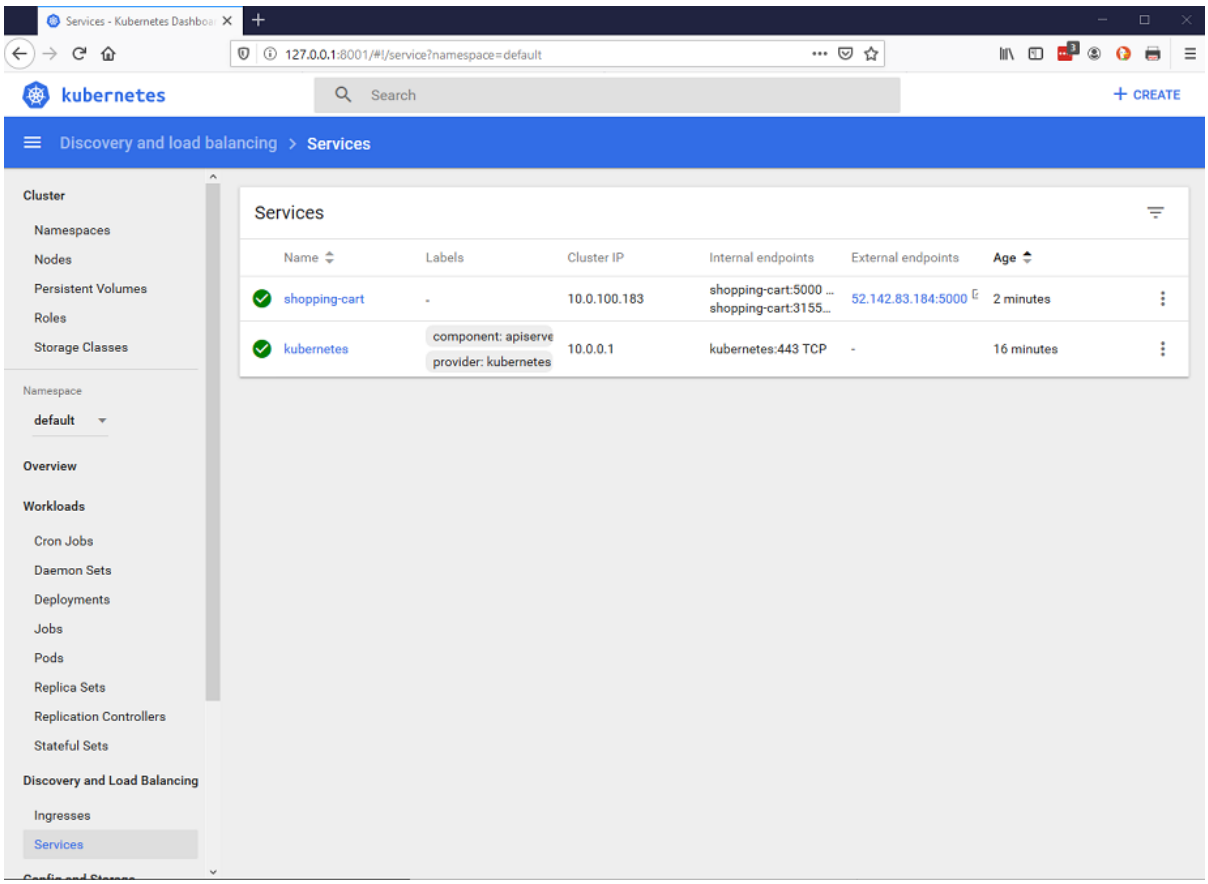

Figure 3.8 Find the address of the Shopping Cart under services

To verify that the Shopping Cart is indeed running we can test it's endpoint by, for instance, adding items to a cart using the Shopping Cart's POST endpoint:

POST http:52.142.83.184:5000///shoppingcart/15/items Accept: application/json Content-Type: applica
2]

and then reading the same cart using the GET endpoint:

GET http:52.142.83.184:5000///shoppingcart/15

The response from this endpoint is the list of items in the cart for user number 15, confirming that the shopping cart works just the same on AKS as it did on Kubernetes on localhost and when we ran the Shopping Cart directly with `dotnet run`.

In a microservices systems we will over time create many microservices which in turn means we will have many pods in Kubernetes, following the pattern I have layed out in this chapter each one has an associated load balancer in Kubernetes too which allows balancing traffic between multiple copies of the containers. For instance we might run 3, 10, or 100 copies of Shopping Cart container and let the Shopping Carts load balancer spread the incoming traffic between them. With a load balancer for each service each exposing endpoints to the outside and each with a different IP address a problem arises: How do we manage all these different entry points into

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

our Kubernetes cluster? There is a security aspect to this question that we will return to in chapter 11 and there is a question of how end user applications know where to send requests which we will return to in chapter 13.

This concludes the setup of a production-like environment and the deployment of the first microservice to that environment. This positions us well for continuing working with microservices. The AKS cluster is setup and ready to run more microservices. As we develop more microservices throughout we will include a `Dockerfile` and a Kubernetes manifest in each one which enables us to build container images, push them to the private registry and deploy them to AKS.

3.5 Summary

- The microservices we develop can be deployed to many different environments
- .NET Core based microservices are easily put into containers and run as containers
- Dockerfiles for .NET Core based microservices follow a common template
- Deploying our microservices to Kubernetes gives us a highly scalable environment and a versatile container orchestrator
- Kubernetes works the same localhost and in the cloud. This means we can easily run the exact same containers localhost for development and in the cloud for production.
- Kubernetes manifests for our microservices are all similar
- Kubernetes can run everything we are going to develop in the upcoming chapters while providing tools for scaling, monitoring, and debugging microservices.
- Azure Kubernetes Services is an easy to setup managed Kubernetes offering that enables us to get up and running with Kubernetes quickly

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

4 Identifying and scoping microservices

This chapter covers

- Scoping microservices for business capability
- Scoping microservices to support technical capabilities
- Scoping microservice to support efficient development work
- Managing when scoping microservices is difficult
- Carving out new microservices from existing ones

To succeed with microservices, it's important to be good at scoping each microservice appropriately. If your microservices are too big, the turnaround on creating new features and implementing bug fixes becomes longer than it needs to. If they're too small, the coupling between microservices tends to grow. If they're the right size but have the wrong boundaries, coupling also tends to grow, and higher coupling also leads to longer turnaround. In other words, if you aren't able to scope your microservices correctly, you'll lose much of the benefit microservices offer. In this chapter, I'll teach you how to find a good scope for each microservice so they stay loosely coupled and efficient to work with.

The primary driver in identifying and scoping microservices is business capabilities, the secondary driver is supporting technical capabilities, and the tertiary driver is efficiency of work. Following these drivers leads to microservices that align nicely with the list of microservice characteristics from chapter 1:

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A microservice consists of one or more processes.
- A microservice owns its own data store.
- A small team can maintain a handful of microservices.
- A microservice is replaceable.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Of these characteristics, the first two and last two can only be realized if the microservice's scope is good. There are also implementation-level concerns that come into play, but getting the scope wrong will prevent the service from adhering to those four characteristics.

4.1 The primary driver for scoping microservices: business capabilities

Each microservice should implement exactly one capability. For example, a Shopping Cart microservice should keep track of the items in the user's shopping cart. The primary way to identify capabilities for microservices is to analyze the business problem and determine the business capabilities. Each business capability should be implemented by a separate microservice.

4.1.1 What is a business capability?

A *business capability* is something an organization does that contributes to business goals. For instance, handling a shopping cart on an e-commerce website is a business capability that contributes to the broader business goal of allowing users to purchase items. A given business will have a number of business capabilities that together make the overall business function.

When mapping a business capability to a microservice, the microservice models the business capability. In some cases, the microservice implements the entire business capability and automates it completely. In other cases, the microservice implements only part of the business capability, because some part of the capability is carried out by people in the real world and thus the microservice only partly automates the business capability. In both cases, the scope of the microservice is the business capability.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Business capabilities and bounded contexts**

Domain-driven design is an approach to designing software systems that's based on modeling the business domain. An important step is identifying the language used by domain experts to talk about the domain. It turns out that the language used by domain experts isn't consistent in all cases.

In different parts of a domain, different things are in focus, so a given word like *customer* may have different focuses in different parts of the domain. For instance, for a company selling photocopiers, a *customer* in the sales department may be a company that buys a number of photocopiers and may be primarily represented by a procurement officer. In the *customer* service department, a customer may be an end user having trouble with a photocopier. When modeling the domain of the photocopier company, the word *customer* means different things in different parts of the model.

A *bounded context* in domain-driven design is a part of a larger domain within which words mean the same things. Bounded contexts are related to but different from business capabilities. A bounded context defines an area of a domain within which the language is consistent. Business capabilities, on the other hand, are about what the business needs to get done. Within one bounded context, the business may need to get several things done. Each of these things is likely a business capability.

4.1.2 Identifying business capabilities

A good understanding of the domain will enable you to understand how the business functions. Understanding how the business functions means you can identify the business capabilities that make up the business and the processes involved in delivering the capabilities. In other words, the way to identify business capabilities is to learn about the business's domain. You can gain this type of knowledge by talking with the people who know the business domain best: business analysts, the end users of your software, and so on—all the people directly involved in the day-to-day work that drives the business.

A business's organization usually reflects its domain. Different parts of the domain are handled by different groups of people, and each group is responsible for delivering certain business capabilities; so, this organization can give you hints about how the microservices should be scoped. For one thing, a microservice's responsibility should probably lie within the purview of only one group. If it crosses the boundary between two groups, it's probably too widely scoped and will be difficult to keep cohesive, leading to low maintainability. These observations are in line with what is known as *Conway's Law*: [10]

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Sometimes you may uncover parts of the domain where the organization and the domain are at odds. In such situations, there are two approaches you can take, both of which respect Conway's Law. You can accept that the system can't fully reflect the domain, and implement a few microservices that aren't well aligned with the domain but are well aligned with the organization; or you can change the organization to reflect the domain. Both approaches can be problematic. The first risks building microservices that are poorly scoped and that might become highly coupled. The second involves moving people and responsibilities between groups. Those kinds of changes can be difficult. Your choice should be a pragmatic one, based on an assessment of which approach will be least troublesome.

To get a better understanding of what business capabilities are, it's time to look at an example.

4.1.3 Example: point-of-sale system

The example we'll explore in this chapter is a point-of-sale system, illustrated in figure 4.1. I'll briefly introduce the domain, and then we'll look at how to identify business capabilities within it. Finally, we'll consider in more detail the scope of one of the microservices in the system.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

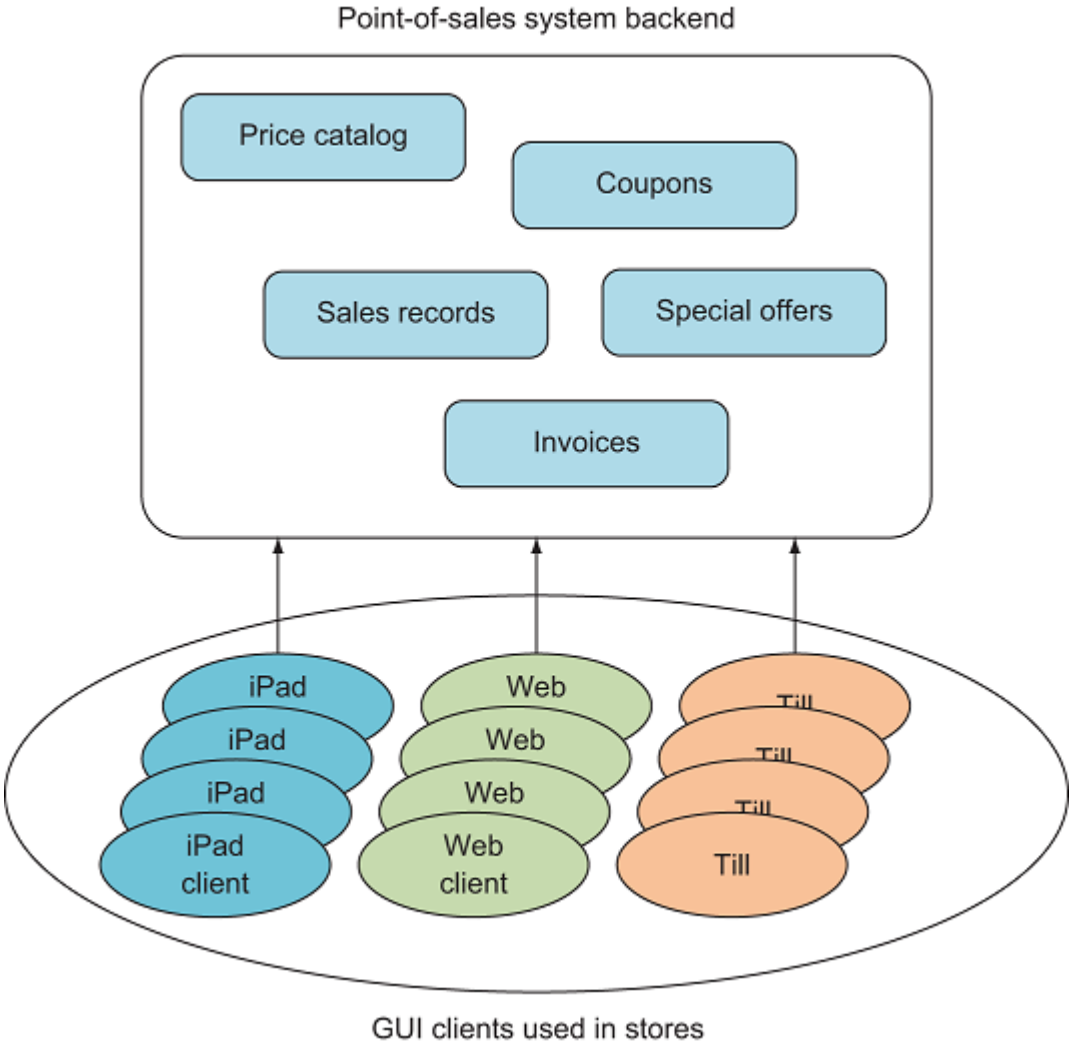

Figure 4.1 A point-of-sale system for a large chain of stores, consisting of a backend that implements all the business capabilities in the system and thin GUI clients used by cashiers in the stores. Microservices in the backend implement the business capabilities.

This point-of-sale system is used in all the stores of a large chain. Cashiers at the stores interact with the system through a thin GUI client—it could be a tablet application, a web application, or a physical purpose-built till (or register, if you prefer). The GUI client is just a thin layer in front of the backend. The backend is where all the business logic (the business capabilities) is implemented, and it will be our focus.

The system offers cashiers a variety of functions:

- Scan products and add them to the invoice
- Prepare an invoice
- Charge a credit card via a card reader attached to the client
- Register a cash payment
- Accept coupons
- Print a receipt

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

- Send an electronic receipt to the customer
- Search in the product catalog
- Scan one or more products to show prices and special offers related to the products

These functions are things the system does for the cashier, but they don't directly match the business capabilities that drive the point-of-sale system.

IDENTIFYING BUSINESS CAPABILITIES IN THE POINT-OF-SALE DOMAIN

To identify the business capabilities that drive the point-of-sale system, you need to look beyond the list of functions. You must determine what needs to go on behind the scenes to support the functionality.

Starting with the "Search in the product catalog" function, an obvious business capability is maintaining a product catalog. This is the first candidate for a business capability that could be the scope of a microservice. Such a Product Catalog microservice would be responsible for providing access to the current product catalog. The product catalog needs to be updated every so often, but the chain of stores uses another system to handle that functionality. The Product Catalog microservice would need to reflect the changes made in that other system, so the scope of the Product Catalog microservice would include receiving updates to the product catalog.

The next business capability you might identify is applying special offers to invoices. Special offers give the customer a discounted price when they buy a bundle of products. A bundle may consist of a certain number of the same product at a discounted price (for example, three for the price of two) or may be a combination of different products (say, buy A and get 10% off B). In either case, the invoice the cashier gets from the point-of-sale GUI client must take any applicable special offers into account automatically. This business capability is the second candidate to be the scope for a microservice. A Special Offers microservice would be responsible for deciding when a special offer applies and what the discount for the customer should be.

Looking over the list of functionality again, notice that the system should allow cashiers to "Scan one or more products to show prices and special offers related to the products." This indicates that there's more to the Special Offers business capability than just applying special offers to invoices: it also includes the ability to look up special offers based on products.

If you continued the hunt for business capabilities in the point-of-sale system, you might end up with this list:

- Product Catalog
- Price Catalog
- Price Calculation
- Special Offers
- Coupons
- Sales Records

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

- Invoice
- Payment

Figure 4.2 shows a map from functionalities to business capabilities. The map is a logical one, in the sense that it shows which business capabilities are needed to implement each function, but it doesn't indicate any direct technical dependencies. For instance, the arrow from Prepare Invoice to Coupons doesn't indicate a direct call from some Prepare Invoice code in a client to a Coupons microservice. Rather, the arrow indicates that in order to prepare an invoice, coupons need to be taken into account, so the Prepare Invoice function depends on the Coupons business capability.

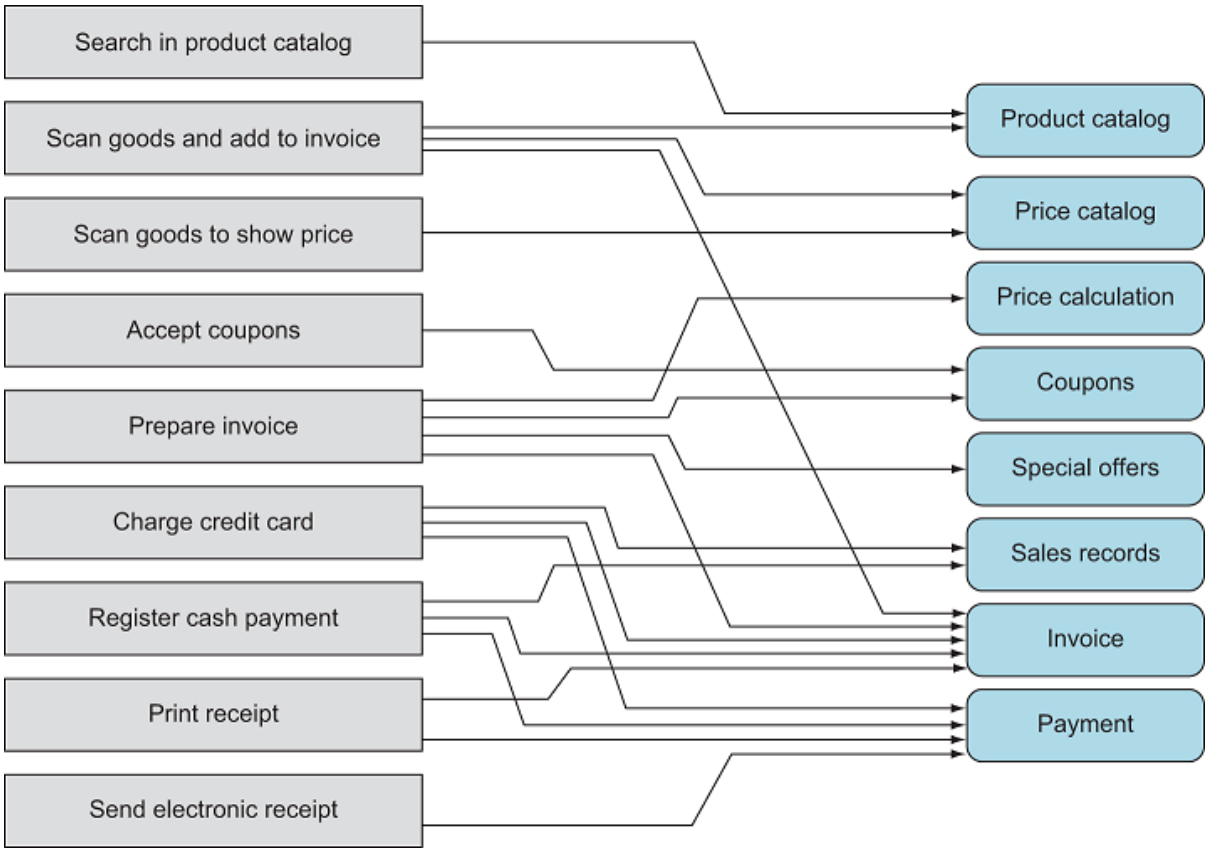

Figure 4.2 The functions on the left depend on the business capabilities on the right. Each arrow indicates a dependency between a function and a capability.

Finding the business capabilities in real domains can be hard work and often requires a good deal of iterating. The list of business capabilities isn't a static list made at the start of development; rather, it's an emergent list that grows and changes over time as your understanding of the domain and the business grows and deepens.

Now that we've gone through the first iteration of identifying business capabilities, let's take a closer look at one of these capabilities and how it defines the scope of a microservice.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

THE SPECIAL OFFERS MICROSERVICE

The Special Offers microservice is based on the Special Offers business capability. To narrow the scope of this microservice, we'll dive deeper into this business capability and identify the processes involved, illustrated in figure 4.3. Each process delivers part of the business capability.

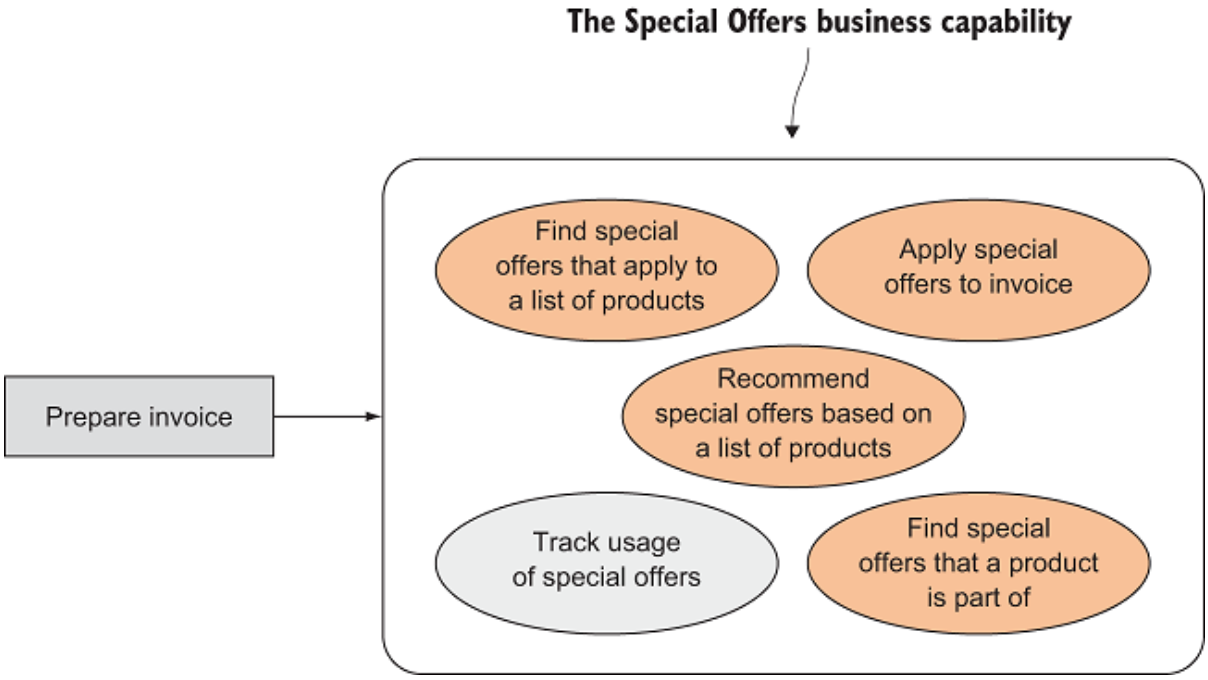

Figure 4.3 The Special Offers business capability includes a number of different processes.

The Special Offers business capability is broken down into five processes. Four of these are oriented toward the point-of-sale GUI clients. The fifth—tracking the use of special offers—is oriented toward the business itself, which has an interest in which special offers customers are taking advantage of.

Implementing the business capability as a microservice means you need to do the following:

- Expose the four client-oriented processes as API endpoints that other microservices can call.
- Implement the usage-tracking process through an event feed. The business-intelligence parts of the point-of-sale system can subscribe to these events and use them to track which special offers are used by customers.

The components of the Special Offers microservice are shown in figure 4.4.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

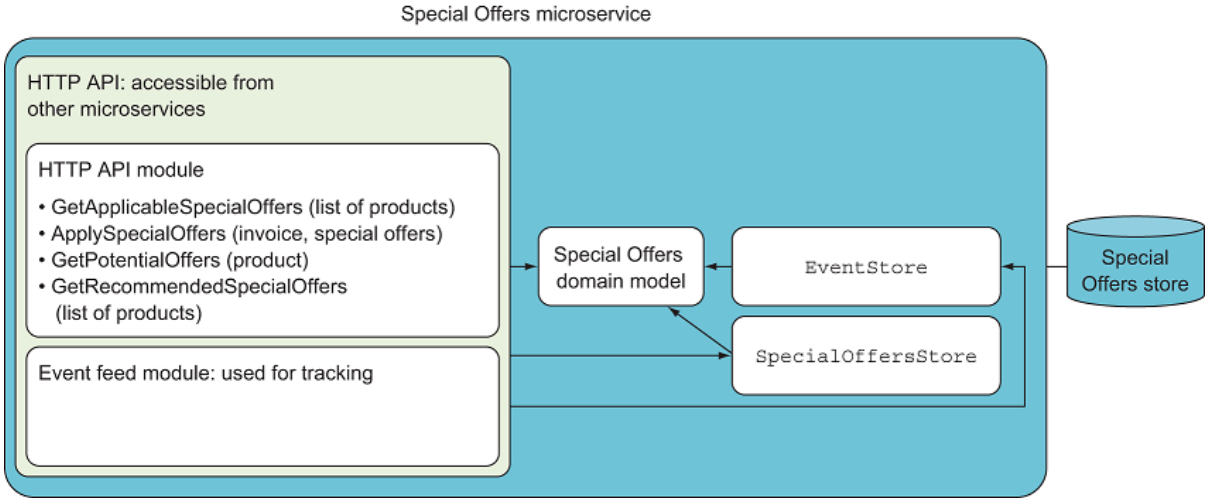

Figure 4.4 The processes in the Special Offers business capability are reflected in the implementation of the Special Offers microservice. The processes are exposed to other microservices through the microservice's HTTP API.

The components of the Special Offers microservice are similar to the components of the Shopping Cart microservice in chapter 2, which is shown again in figure 2.3. This is no coincidence. These are the components our microservices typically consist of: an HTTP API that exposes the business capability implemented by the microservice, an event feed, a domain model implementing the business logic involved in the business capability, a data store component, and a database.

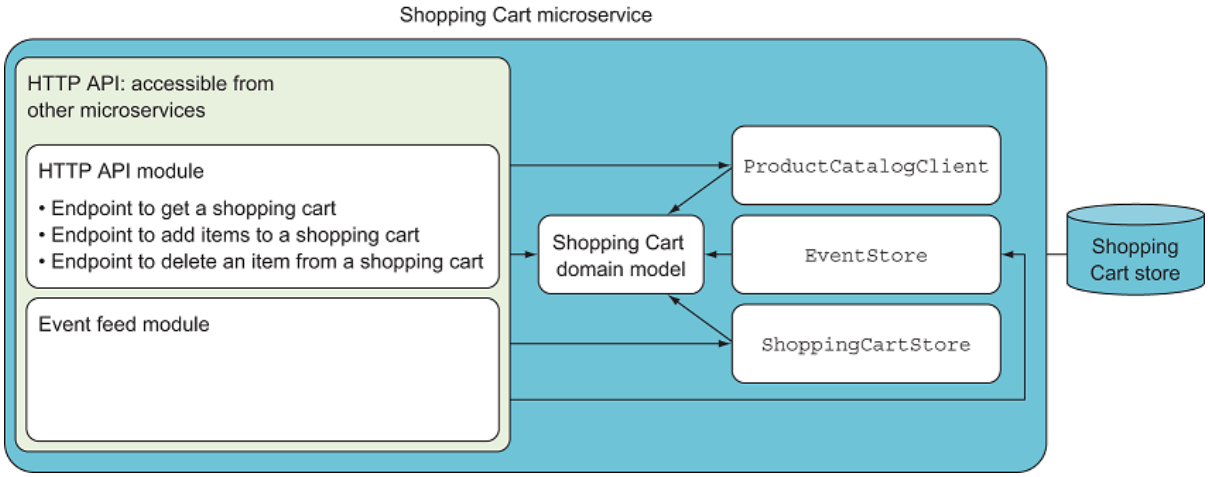

Figure 4.5 The components of the Shopping Cart microservice from chapter 2 are similar to the components of the Special Offers microservice.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

4.2 The secondary driver for scoping microservices: supporting technical capabilities

The secondary way to identify scopes for microservices is to look at supporting technical capabilities. A *supporting technical capability* is something that doesn't directly contribute to a business goal but supports other microservices, such as integrating with another system or scheduling an event to happen some time in the future.

4.2.1 What is a technical capability?

Supporting technical capabilities are a secondary driver in scoping microservices because they don't directly contribute to the system's business goals. They exist to simplify and support the other microservices that implement business capabilities.

Remember, one characteristic of a good microservice is that it's replaceable; but if a microservice that implements a business capability also implements a complex technical capability, it may grow too large and too complex to be replaceable. In such cases, you should consider implementing the technical capability in a separate microservice that supports the original one. Before discussing how and when to identify supporting technical capabilities, a couple of examples would probably be helpful.

4.2.2 Examples of supporting technical capabilities

To give you a feel for what I mean by supporting technical capabilities, let's consider two examples: an integration with another system, and the ability to send notifications to customers.

INTEGRATING WITH AN EXTERNAL PRODUCT CATALOG SYSTEM

In the example point-of-sale system, you identified the product catalog as a business capability. I also mentioned that product information is maintained in another system, external to the microservice-based point-of-sale system. That other system is an Enterprise Resource Planning (ERP) system. This implies that the Product Catalog microservice must integrate with the ERP system, as illustrated in figure 4.6.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

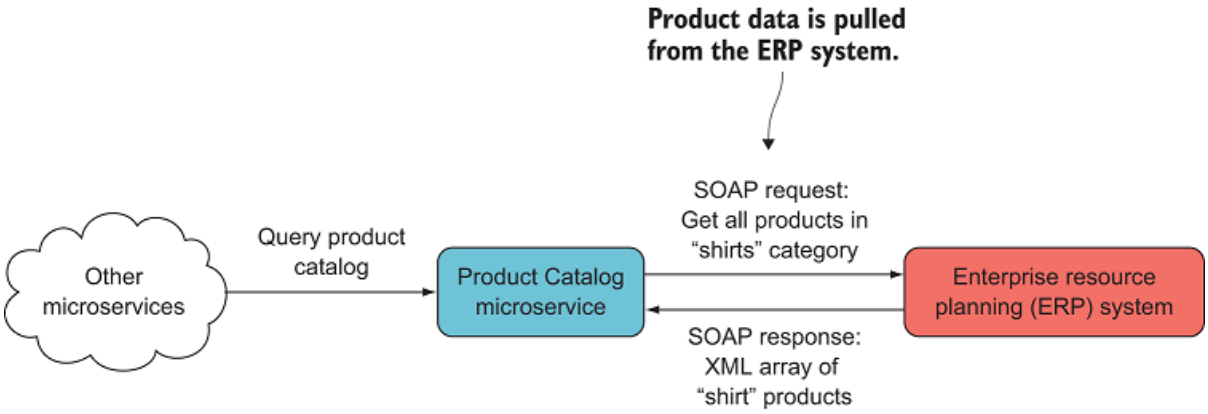

Figure 4.6 Product data flows from the ERP system to the Product Catalog microservice. The protocol used to get product information from the ERP system is defined by the ERP system. It could expose a SOAP web service for fetching the information, or it might export product information to a proprietary file format.

Let's assume that you aren't in a position to make changes to the ERP system, so the integration must be implemented using whatever interface the ERP system has. It might use a SOAP web service to fetch product information, or it might export all the product information to a proprietary file format. In either case, the integration must happen on the ERP system's terms. Depending on the interface the ERP system exposes, this may be a smaller or larger task. In any case, it's a task primarily concerned with the technicalities of integrating with some other system, and it has the potential to be at least somewhat complex. The purpose of this integration is to support the Product Catalog microservice.

We'll take the integration out of the Product Catalog microservice and implement it in a separate ERP Integration microservice that's responsible solely for that one integration, as illustrated in figure 4.7. We'll do this for two reasons:

- By moving the technical complexities of the integration to a separate microservice, we keep the scope of the Product Catalog microservice narrow and focused.
- By using a separate microservice to deal with how the ERP data is formatted and organized, we keep the ERP system's view of what a product is separate from the point-of-sale system. Remember that in different parts of a large domain, there are different views of what terms mean. It's unlikely that the Product Catalog microservice and the ERP system agree on how the product entity is modeled. A translation between the two views is needed and is best done by the new microservice. In domain-driven-design terms, the new microservice acts as an *anti-corruption layer*.

NOTE The anti-corruption layer is a concept borrowed from domain-driven design. It can be used when two systems interact; it protects the domain model in one system from being polluted with language or concepts from the model in the other system.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

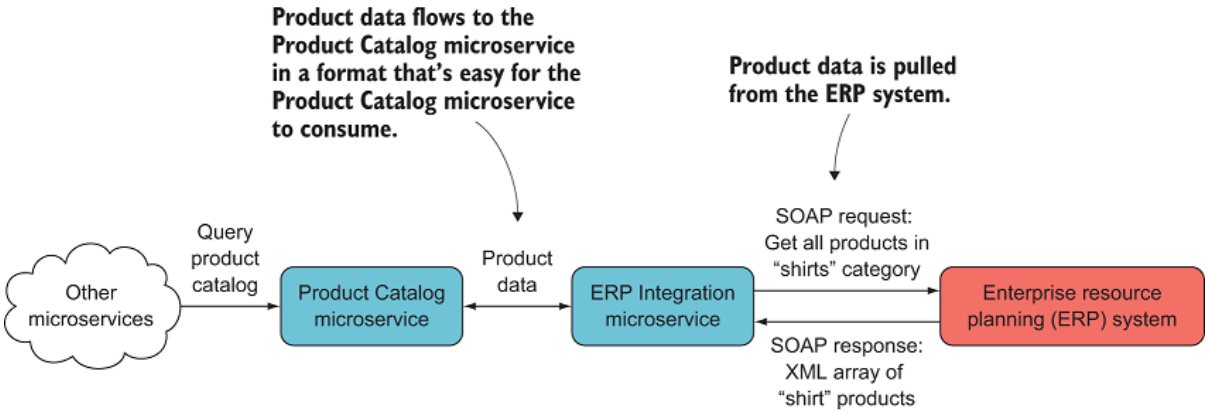

Figure 4.7 The ERP Integration microservice supports the Product Catalog microservice by handling the integration with the ERP system. It translates between the way the ERP system exposes product data and the way the Product Catalog microservice consumes it.

An added benefit of placing the integration in a separate microservice is that it's a good place to address any reliability issues related to the integration. If the ERP system is unreliable, the place to handle that is in the ERP Integration microservice. If the ERP system is slow, the ERP Integration microservice can deal with that. Over time, you can tweak the policies used in the ERP Integration microservice to address any reliability issues with the ERP system without touching the Product Catalog microservice at all. This integration with the ERP system is an example of a supporting technical capability, and the ERP Integration microservice is an example of a microservice implementing that capability.

SENDING NOTIFICATIONS TO CUSTOMERS

Now let's consider extending the point-of-sale system with the ability to send notifications about new special offers to registered customers via email, SMS, or push notification to a mobile app. We can put this capability into one or more separate microservices.

At the moment, the point-of-sale system doesn't know who the customers are. To drive better customer engagement and customer loyalty, the company decides to start a small loyalty program where customers can sign up to be notified about special offers. The customer loyalty program is a new business capability and will be the responsibility of a new Loyalty Program microservice. Figure 4.8 shows this microservice, which is responsible for notifying registered customers every time a new special offer is available to them.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

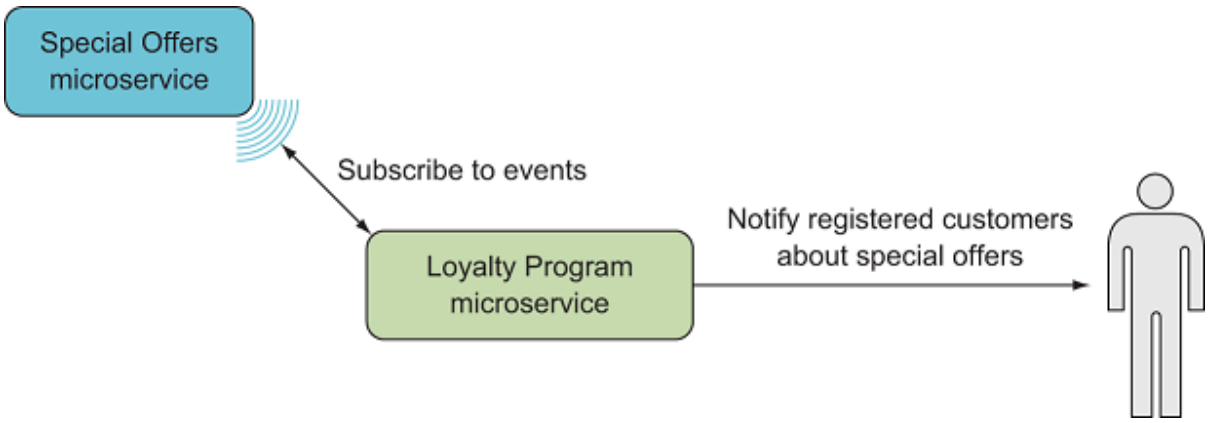

Figure 4.8 The Loyalty Program microservice subscribes to events from the Special Offers microservice and notifies registered customers when new offers are available.

As part of the registration process, customers can choose to be notified by email, SMS, or, if they have the company's mobile app, push notification. This introduces some complexity in the Loyalty Program microservice in that it must not only choose which type of notification to use but also deal with how each one works. As a first step, we'll introduce a supporting technical microservice for each notification type.

This is shown in figure 4.9.

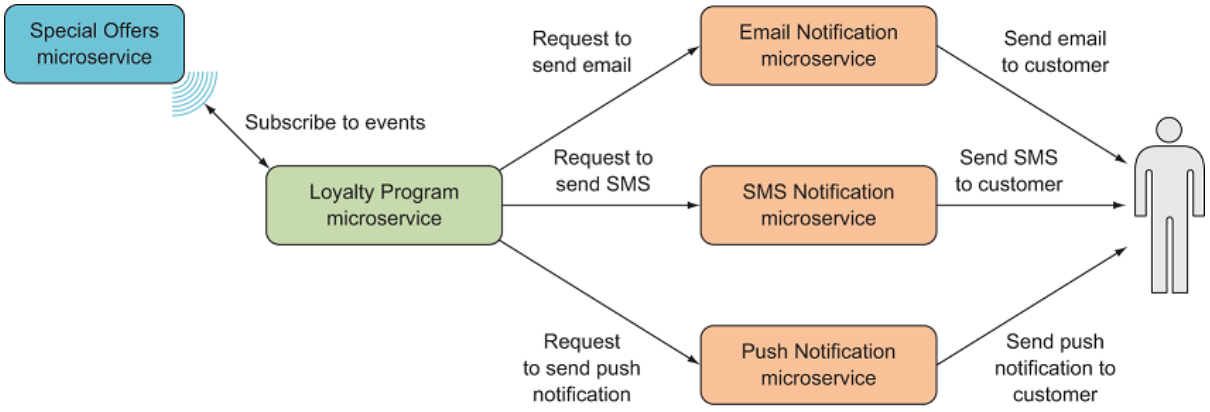

Figure 4.9 To avoid bogging down the Loyalty Program microservice in technical details for handling each type of notification, we'll introduce three supporting technical microservices, one for each type of notification.

This is better. The Loyalty Program microservice doesn't have to implement all the details of dealing with each type of notification, which keeps the microservice's scope narrow and focused. The situation isn't perfect, though: the microservice still has to decide which of the supporting technical microservices to call for each registered customer.

This leads us to introducing one more microservice, which acts as a front for the three microservices handling the three types of notifications. This new Notifications microservice is depicted in figure 4.10 and is responsible for choosing which type of notification to use each time a customer needs to be notified.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

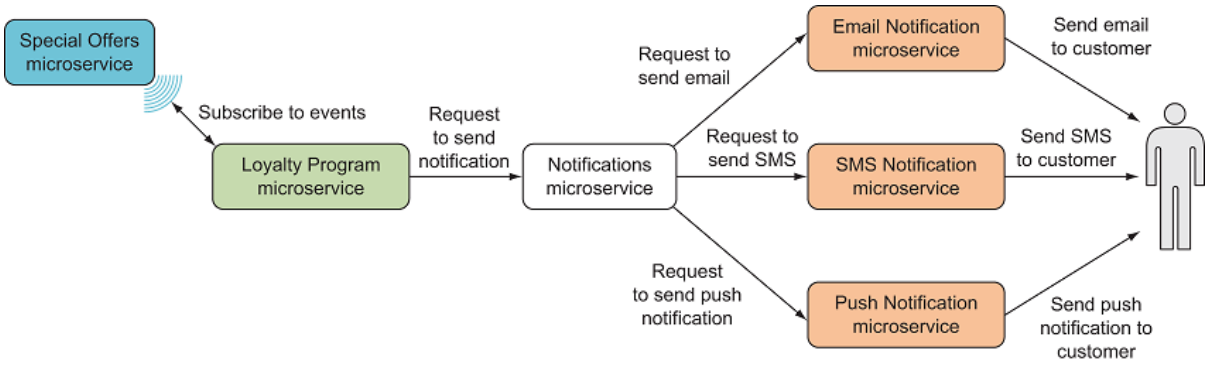

Figure 4.10 To remove more complexity from the Loyalty Program microservice, we'll introduce a Notifications microservice that's responsible for choosing a type of notification based on customer preferences. Introducing this microservice has the added benefit of making notifications easier to use from other microservices.

This example of a supporting technical capability differs from the previous example of the ERP integration in that other microservices may also need to send notifications to specific customers. For instance, one of the functionalities of the point-of-sales system is to send the customer an electronic receipt. The microservice in charge of that business capability can also take advantage of the Notifications microservice. Part of the motivation for moving this to a separate microservices is that you can reuse the implementation.

While sending emails, text messages or push notifications are technical capabilities sending notifications is likely a business capability with business rules and how and when to notify customers. Thus we have discovered a separate business capability by driving out the technical capabilities into their own microservices. This is not not unusual: In practice the three drivers for scoping microservices interact and driving along one axis - in this case the axis of technical capabilities - exposes things along the other axes - in this case the axis of business capabilities. This realization leads us to revise our understanding of the domain and introduce a *Notifications* bounded context that contains the business capability of notifying users as well as the technical capabilities of sending emails, SMS's and push notifications. This new bounded context and the microservices in it is shown in figure 4.11

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

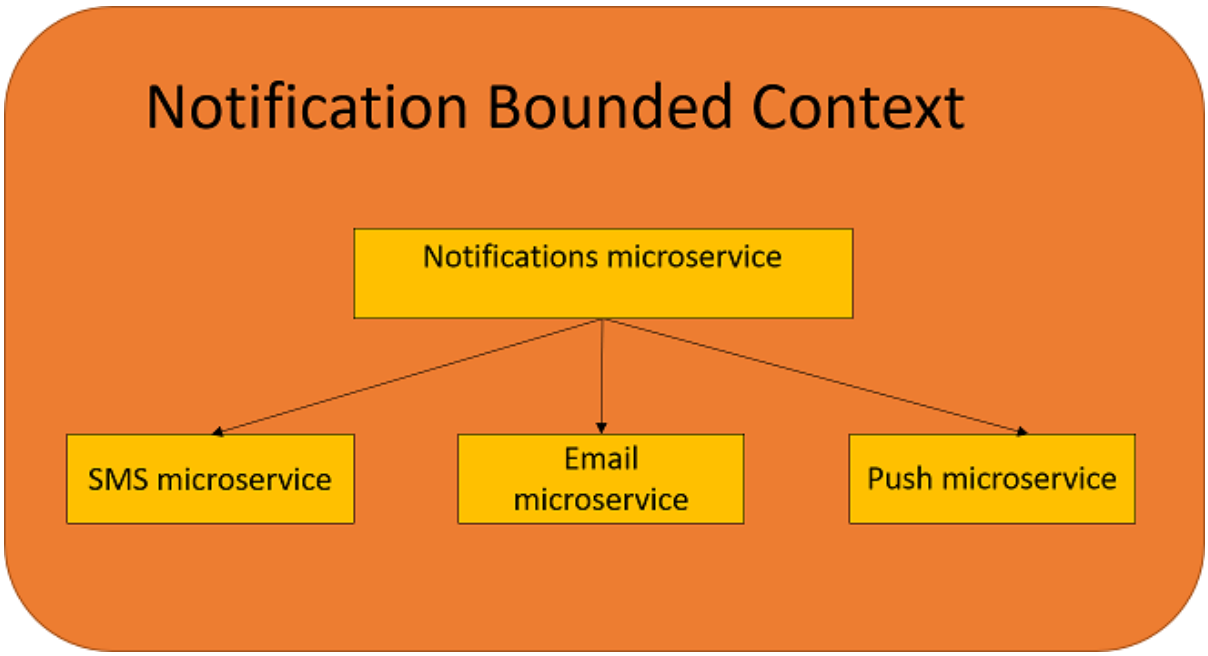

Figure 4.11 The new Notifications bounded context contains the business capability of handling notifications and the technical capabilities of sending emails, SMS's, and push notifications.

The notifications microservice is different from the other three microservices in the notifications bounded context because it handles the business rules around notifications such as respecting customers preferences about how they want to be notified, when they should be notified, how often they should be notified and so on. For instance, the notification microservice might implement rules that adjust the time of day customers are notified about special offers based on their purchasing patterns - so we send them notifications just when they are about to go shopping anyway. Another rule could be that we do not notify any customer about more than three special offers a day to avoid annoying them.

4.2.3 Identifying technical capabilities

When you introduce supporting technical microservices, your goal is to simplify the microservices that implement business capabilities. Sometimes—such as with sending notifications—you identify a set of technical capabilities that several microservices need, and you turn that into a bounded context with microservices of its own, so other microservices can share the implementation. Other times—as with the ERP integration—you identify a technical capability that unduly complicates a microservice and turn that capability into a microservice of its own. In both cases, the other microservices implementing business capabilities are left with one less technical concern to take care of.

When deciding to implement a technical capability in a separate microservice, be careful that you don't violate the microservice characteristic of being individually deployable. It makes sense to implement a technical capability in a separate microservice only if that microservice can be deployed and redeployed independently of any other microservices. Likewise, deploying the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

microservices that are supported by the microservice providing the technical capability must not force you to redeploy the microservice implementing the technical capability.

Identifying business capabilities and microservices based on business capabilities is a strategic exercise, but identifying technical supporting capabilities that could be implemented by separate microservices is an opportunistic exercise. The question of whether a supporting technical capability should be implemented in its own microservice is about what will be easiest in the long run. You should ask these questions:

- If the supporting technical capability stays in a microservice scoped to a business capability, is there a risk that the microservice will no longer be replaceable with reasonable effort?
- Is the supporting technical capability implemented in several microservices scoped to business capabilities?
- Will a microservice implementing the supporting capability be individually deployable?
- Will all microservices scoped to business capabilities still be individually deployable if the supporting technical capability is implemented in a separate microservice?

If your answer is “Yes” to the last two questions and to at least one of the others, you have a good candidate for a microservice scope.

4.3 The tertiary driver for scoping microservices: supporting efficiency of work

The third driver when we find the scope and boundaries of microservices is supporting efficiency of work: The microservices we design and created should be efficient to work with, and should let teams that develop and maintain the microservices work efficiently. This, largely, is matter of respecting and using Conway’s law which you may re-call from earlier is:

> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure.*

If we have an already-defined organization that we cannot change the system of microservices we create must align with the organization. On the other hand if we organization can be changed we should do so in accordance with the overall system of microservices.

First and foremost respecting Conway’s law means that we should make sure that there is a clear ownership of each microservices: We should assign one team to the development and maintenance of each microservice. This is in line with the microservices characteristic that a small team can maintain a few handfuls of microservices. Each can be be assigned a few handfuls of services to be responsible for. It is important that the responsibility is clear. If the responsibility for a microservice is split between two or more teams we introduce a need for those teams to coordinate closely whenever changes to that microservices need to happen. That coordination takes time and means that none of the teams involved can work autonomously. This

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

is a form of in-efficiency, and it is something we can avoid if we make sure each microservices is assigned to one team.

Allowing teams to work autonomously plays into defining the scope and boundary of the microservices: First, the team must have the skills and knowledge necessary. For instance a team that gets assigned the Notifications microservice from earlier in this chapter must have the understanding of the business rules related to Notifying users.

Second, the teams must be able to work without too much and too detailed coordination with other teams.For example when a team is assigned the Notification microservice they will probably also be assigned the supporting technical microservices for sending SMS's, emails, and push notifications because there is certain level of coupling between those and the Notifications microservices. Because of that coupling assigning the supporting technical microservices to other teams introduces a need for coordination and makes the team less autonomous, so we will probably assigning them all to the same team which therefore takes responsibility for the whole Notifications bounded context. Once they take on the supporting technical microservices the team on top of the knowledge about the business rules around Notifications the team need to have the skills necessary to create the integrations to the SMS's gateway, the email system, and to send push notifications. We need to be aware of this, and if we cannot find or form a team the necessary combination of knowledge and skills we may have to change scopes of our microservices.

To make sure we do not create inefficiencies for the teams developing the microservices it pays to keep and eye on the dependency chains that exists in the system microservices. In figure 4.12 a dependency chain is depicted - the boxes are microservices and the arrows are dependencies.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

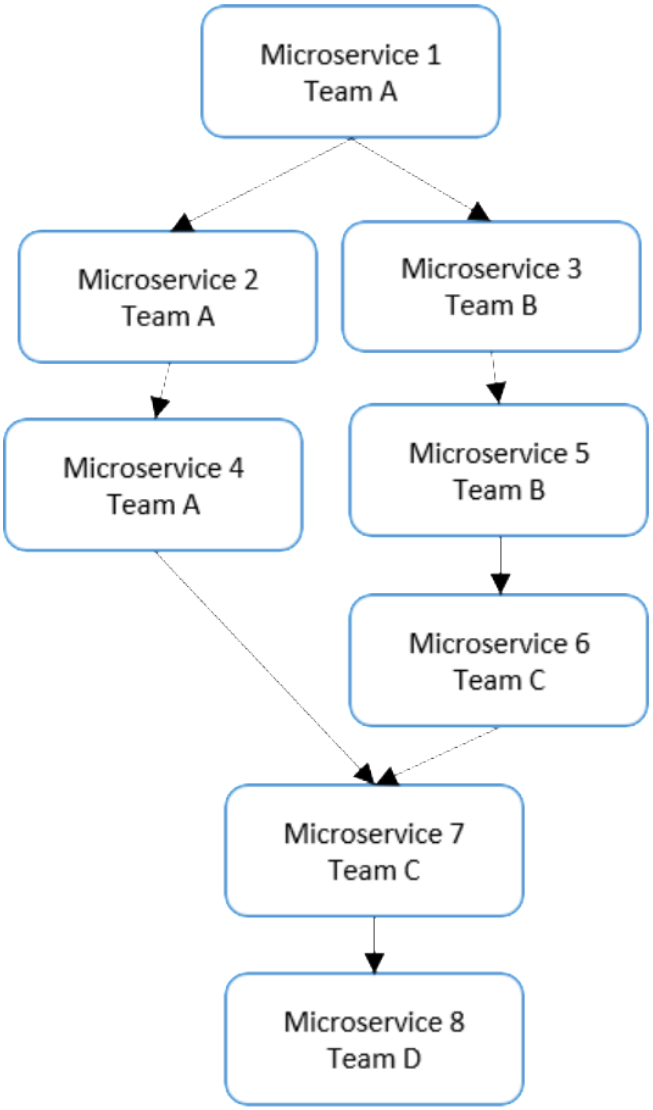

Figure 4.12 Dependencies between microservices are mirrored as dependencies between the team maintaining the microservices. The arrows are dependencies and the boxes are microservices annotated with the teams maintaining them.

If the each microservices in figure 4.12 are assigned to teams A, B, C, and D as indicated inside the boxes the team assigned to the microservice at top of the chain - Team A - is in a very different situation than the team assigned to microservice at the bottom of the chain - Team D. This is because the dependencies between the microservices is mirrored by dependencies between the teams. Team A maintaining the microservices at the top can change anything they want without ever thinking about breaking anything for other teams maintaining microservices further down the chain. Team D maintaining the microservice at the bottom on the other hand risks breaking all the other microservices in the dependency chain if they introduce a breaking change. At the same time Team A maintaining the microservice at the top may not be able get

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

much done without coordinating with the other teams in the dependency chain because they may depend on the other teams to implement stuff. In other words we may be setting both Team A and Team B up for inefficiency if we create too long dependency chains. This is in part combatted by getting the scopes and boundaries for microservices right and in part by designing the collaboration between microservice in a way the favors low coupling. We will go in depth with designing the collaboration between microservice in chapter 5.

> **NOTE** In chapter 5 we will see how even based collaboration can help break dependency chains

4.4 What to do when the correct scope isn't clear

At this point, you may be thinking that scoping microservices correctly is difficult: you need to get the business capabilities just right, which requires a deep understanding of the business domain, you also have to judge the complexity of supporting technical capabilities correctly, and you need to make sure the teams can work efficiently. And you're right: it is difficult, and you *will* find yourself in situations where the right scoping for your microservices isn't clear.

This lack of clarity can have several causes, including the following:

- *Insufficient understanding of the business domain*—Analyzing a business domain and building up a deep knowledge of that domain is difficult and time consuming. You'll sometimes need to make decisions about the scope of microservices before you've been able to develop sufficient understanding of the business to be certain you're making the correct decisions.
- *Confusion in the business domain*—It's not only the development side that can be unclear about the business domain. Sometimes the business side is also unclear about how the business domain should be approached. Maybe the business is moving into new markets and must learn a new domain along the way. Other times, the existing business market is changing because of what competitors are doing or what the business itself is doing. Either way, on both the business side and the development side, the business domain is ever-changing, and your understanding of it is emergent.
- *Incomplete knowledge of the details of a technical capability*—You may not have access to all the information about what it takes to implement a technical capability. For instance, you may need to integrate with a badly documented system, in which case you'll only know how to implement the integration once you're finished.
- *Inability to estimate the complexity of a technical capability*—If you haven't previously implemented a similar technical capability, it can be difficult to estimate how complex the implementation of that capability will be.

None of these problems means you've failed. They're all situations that occur time and again. The trick is to know how to move forward in spite of the lack of clarity. In this section, I'll discuss what to do when you're in doubt.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

4.4.1 Starting a bit bigger

When in doubt about the scope of a microservice, it's best to err on the side of making the microservice's scope bigger than it would be ideally. This may sound weird—I've talked a lot about creating small, narrowly focused microservices and about the benefits that come from keeping microservices small. And it's true that significant benefits can be gained from keeping microservices small and narrowly focused. But you must also look at what happens if you err on the side of too narrow a scope.

Consider the Special Offers microservice discussed earlier in this chapter. It implements the Special Offers business capability in a point-of-sale system and includes five different business processes, as illustrated in figure 3.3 and reproduced on the left side of figure 4.13. If you were uncertain about the boundaries of the Special Offers business capability and chose to err on the side of too small a scope, you might split the business capability as shown on the right side of figure 4.13.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

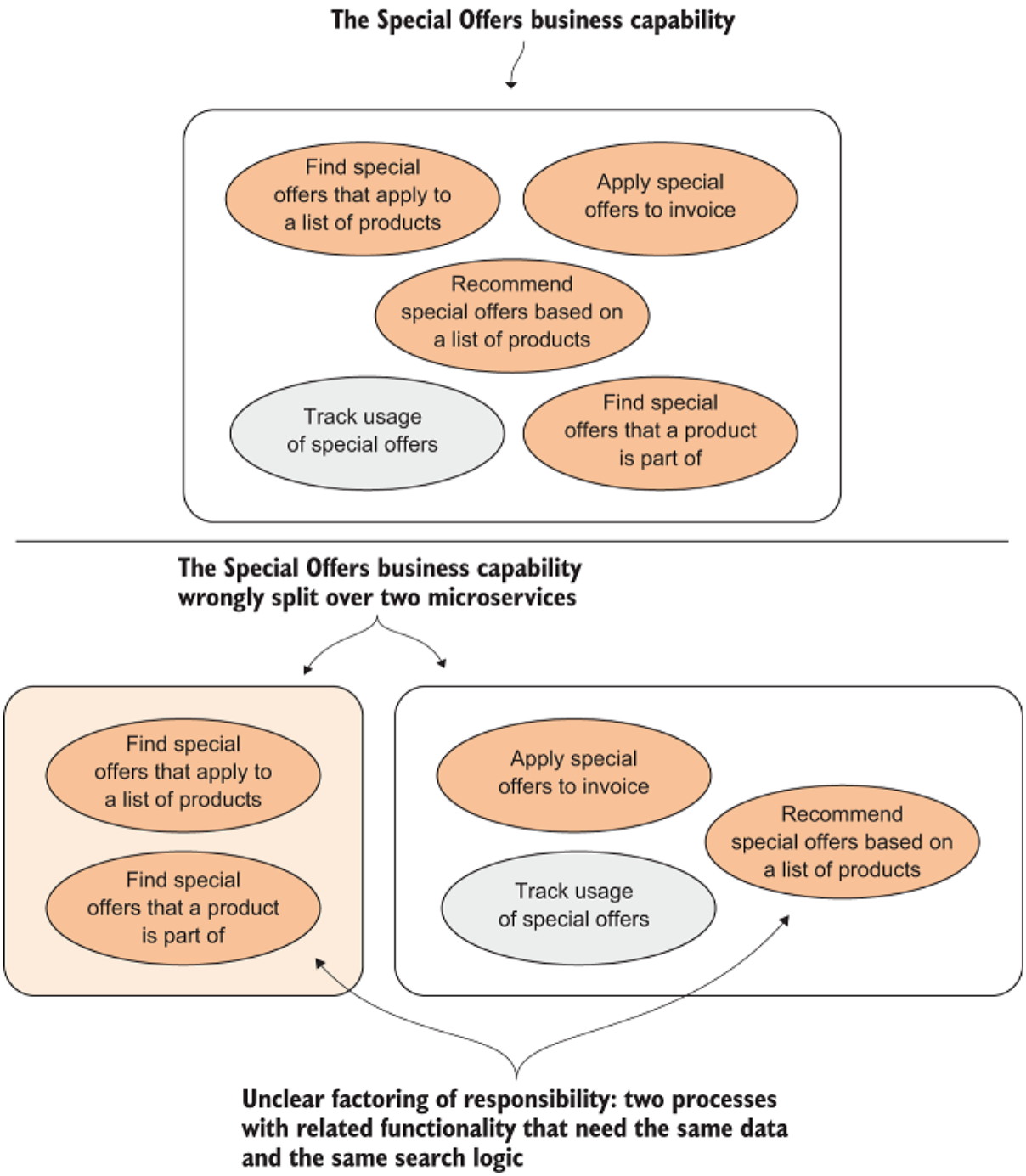

Figure 4.13 If you make the scope of a microservice too small, you'll find that a single business capability becomes split over several highly coupled parts.

If you base the scope of your microservices on only part of the Special Offers business capability, you'll incur some significant costs:

- *Data and data-model duplication between the two microservices*—Both parts of the implementation need to store all the special offers in their data stores.
- *Unclear factoring of responsibility*—One part of the divided business capability can answer whether a given product is part of any special offers, whereas the other part can

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

recommend special offers to customers based on past purchases. These two functions are closely related, and you'll quickly get into a situation where it's unclear in which microservice a piece of code belongs.

- *Obstacles to refactoring the code for the business capability*—This can occur because the code is spread across the code bases for the two microservices. Such cross-code base refactorings are difficult because it's hard to get a complete picture of the consequences of the refactoring and because tooling support is poor.
- *Difficulty deploying the two microservices independently*—After refactoring or implementing a feature that involves both microservices, the two microservices may need to be deployed at the same time or in a particular order. Either way, coupling between versions of the two microservices violates the characteristic of microservices being individually deployable. This makes testing, deployment, and production monitoring more complicated.

These costs are incurred from the time the microservices are first created until you've gained enough experience and knowledge to more correctly identify the business capability and a better scope for a microservice (the entire Special Offers business capability, in this case). Added to those costs is the fact that difficulty refactoring and implementing changes to the business capability will result in you doing less of both, so it will take you longer to learn about the business capability. In the meantime, you pay the cost of the duplicated data and data model and the cost of the lack of individual deployability.

We've established that preferring to err on the side of too narrow a scope easily leads to scoping microservices in a way that creates costly coupling between the microservices. To see if this is better or worse than erring on the side of too big a scope, we need to look at the costs of that approach.

If you err on the side of bigger scopes, you might decide on a scope for the Special Offers microservice that also includes handling coupons. The scope of this bigger Special Offers microservice is shown in figure 4.14.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

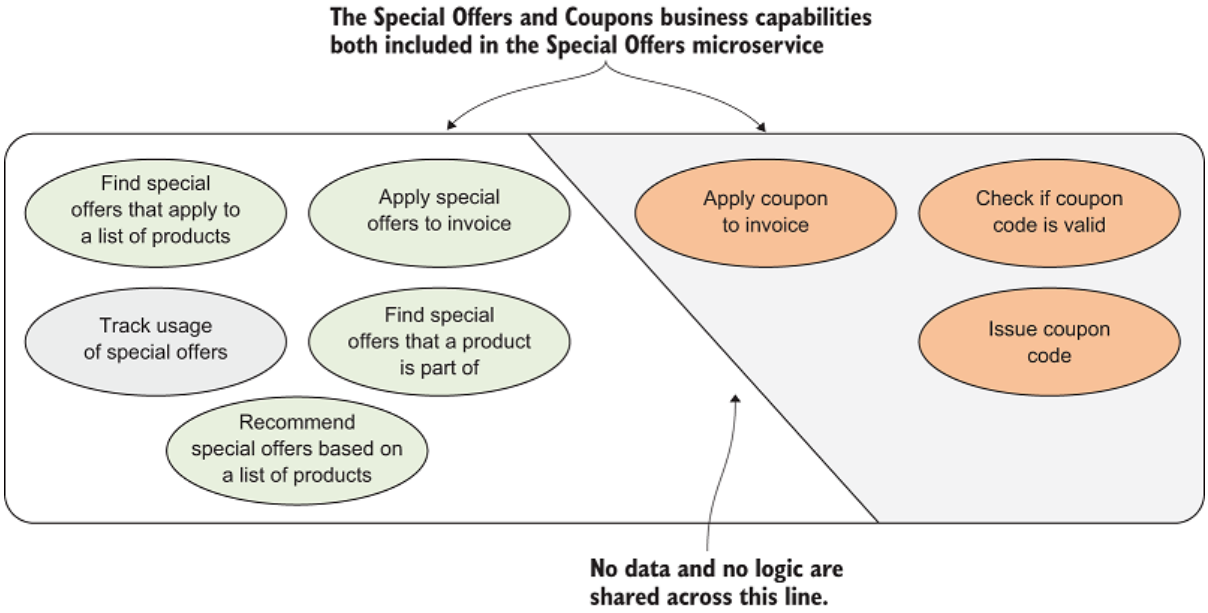

Figure 4.14 If you choose to err on the side of bigger scopes, you might decide to include the handling of coupons in the Special Offers business capability.

There are costs associated with including too much in the scope of a microservice:

- The code base becomes bigger and more complex, which can lead to changes being more expensive.
- The microservice is harder to replace.

These costs are real, but they aren't overwhelming when the scope of the microservice is still fairly small. Beware, though, because these costs grow quickly with the size of each microservice's scope and become overwhelming when the scope is so big that it approaches a monolithic architecture.

Nevertheless, refactoring within one code base is much easier than refactoring across two code bases. This gives you a better chance to experiment and to learn about the business capability through experiments. If you take advantage of this opportunity, you can arrive at a good understanding of both the Special Offers business capability and the Coupons business capability more quickly than if you scoped your microservices too narrowly.

This argument holds true when your microservices are a bit too big, but it falls apart if they're much too big in which case we will loose the speed and flexibility that we want from microservices.

WARNING Don't get lazy and lump several business capabilities together in one microservice. You'll quickly have a large, hard-to-manage code base with many of the drawbacks of a full-on monolith.

All in all, microservices that are slightly bigger than they should ideally be are both less costly

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

and allow for more agility than if they're slightly smaller than they should ideally be. Thus, the rule of thumb is to err on the side of slightly bigger scopes.

Once you accept that you'll sometimes—if not often—be in doubt about the best scope for a microservice and that in such cases you should lean toward a slightly bigger scope, you can also accept that you'll sometimes—if not often—have microservices in your system that are somewhat larger than they should ideally be. This means you should expect to have to carve new microservices out of existing ones from time to time.

4.4.2 Carving out new microservices from existing microservices

When you realize that one of your microservices is too big, you'll need to look at how to carve a new microservice out of it. First you need to identify a good scope for both the existing microservice and the new microservice. To do this, you can use the drivers described earlier in this chapter.

Once you've identified the scopes, you must look at the code to see if the way it's organized aligns with the new scopes. If not, you should begin refactoring toward that alignment. Figure 4.15 illustrates on a high level the refactorings needed to prepare to carve out a new microservice from an existing one. First, everything that will eventually go into the new microservice is moved to its own class library. Then, all communication between code that will stay in the existing microservice and code that will be moved to the new microservice is refactored to go through an interface. This interface will become part of the public HTTP interface of the two microservices once they're split apart.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

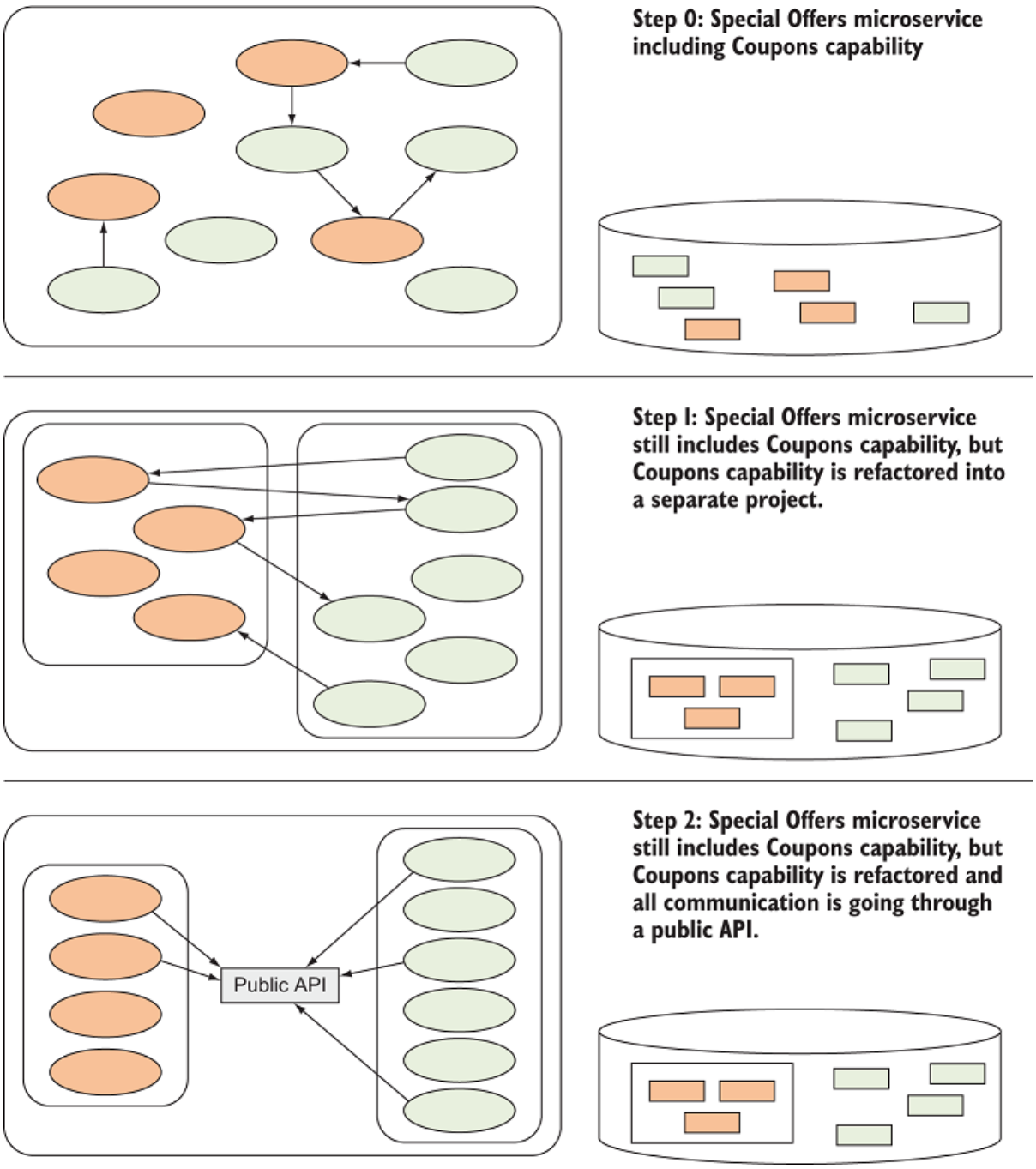

Figure 4.15 Preparing to carve out a new microservice by refactoring: first move everything belonging to the new microservice into its own project, and then make all communication go through a public API similar to the one the new microservice will end up having.

When you've reached step 2 in figure 3.13, the new microservice can be split out from the old one with a manageable effort. Create a new microservice, move the code that needs to be carved out of the existing microservice over to the new microservice, and change the communication between the two parts to go over HTTP.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

4.4.3 Planning to carve out new microservices later

Because you consciously err on the side of making your microservices a bit too big when you're in doubt about the scope of a microservice, you have a chance to foresee which microservices will have to be divided at some point. If you know a microservice is likely to be split later, it would be nice if you could plan for that split in a way that will save you one or two of the refactoring steps shown in figure 3.13. It turns out you can often make that kind of plan.

Often you'll be unsure whether a particular function is a separate business capability, so you'll follow the rule of thumb and include it in a larger business capability, implemented within a microservice scoped to that larger business capability. But you can remain conscious of the fact that this area *might* be a separate business capability.

Think about the definition of the Special Offers business capability that includes processes for dealing with coupons. You may well have been in doubt about whether handling coupons was a business capability on its own, so the Special Offers business capability was modeled as including all the processes shown in figure 3.12.

When you first implement a Special Offers microservice scoped to the understanding of the Special Offers business capability illustrated in figure 3.12, you don't know whether the coupons functionality will eventually be moved to a Coupons microservice. You do know, however, that the coupons functionality isn't as closely related to the rest of the microservice as some of the other areas. It's therefore a good idea to put a clear boundary around the coupons code in the form a well-defined public API and to put the coupons code in a separate class library. This is sound software design, and it will also pay off if one day you end up carving out the coupons code to create a new Coupons microservice.

4.5 Well-scoped microservices adhere to the microservice characteristics

I've talked about scoping microservices by identifying business capabilities first and supporting technical capabilities second. In this section, I'll discuss how this approach to scoping aligns with these four characteristics of microservices mentioned at the beginning of this chapter:

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A small team can maintain a handful of microservices.
- A microservice is replaceable.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

NOTE It's important to note that the relationship between the drivers for scoping microservices and the characteristics of microservices goes both ways. The primary and secondary drivers lead toward adhering to the characteristics, but the characteristics also tell you whether you've scoped your microservices well or need to push the drivers further to find better scopes for your microservices.

4.5.1 Primarily scoping to business capabilities leads to good microservices

The primary driver for scoping microservices is identifying business capabilities. Let's see how that makes for microservices that adhere to the microservice characteristics.

RESPONSIBLE FOR A SINGLE CAPABILITY

A microservice scoped to a single business capability by definition adheres to the first microservice characteristic: it's responsible for a single capability. As you saw in the examples of identifying supporting technical capabilities, you have to be careful: it's easy to let too much responsibility slip into a microservice scoped to a business capability. You have to be diligent in making sure that what a microservice implements is just one business capability and not a mix of two or more. You also have to be careful about putting supporting technical capabilities in their own microservices. As long as you're diligent, microservices scoped to a single business capability adhere to the first characteristic of microservices.

INDIVIDUALLY DEPLOYABLE

Business capabilities are those that can be performed by largely independent groups within an organization, so the business capabilities themselves must be largely independent. As a result, microservices scoped to business capabilities are largely independent. This doesn't mean there's no interaction between such microservices—there can be a lot of interaction, both through direct calls between services and through events. The point is that the interaction happens through well-defined public interfaces that can be kept backward compatible. If implemented well, the interaction is such that other microservices continue to work even if one has a short outage. This means well-implemented microservices scoped to business capabilities are individually deployable.

REPLACEABLE AND MAINTAINABLE BY A SMALL TEAM

A business capability is something a small group in an organization can handle. This limits its scope and thus also limits the scope of microservices scoped to business capabilities. Again, if you're diligent about making sure a microservice handles only one business capability and that supporting technical capabilities are implemented in their own microservices, the microservices' scope will be small enough that a small team can maintain at least a handful of microservices and a microservice can be replaced fairly quickly if need be.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

4.5.2 Secondarily scoping to supporting technical capabilities leads to good microservices

The secondary driver for scoping microservices is identifying supporting technical capabilities. Let's see how that makes for microservices that adhere to the microservice characteristics.

RESPONSIBLE FOR A SINGLE CAPABILITY

Just as with microservices scoped to business capabilities, scoping a microservice to a single supporting technical capability by definition means it adheres to the first characteristic of microservices: it's responsible for a single capability.

INDIVIDUALLY DEPLOYABLE

Before you decide to implement a technical capability as a separate supporting technical capability in a separate microservice, you need to ask whether that new microservice will be individually deployable. If the answer is "No," you shouldn't implement it in a separate microservice. Again, by definition, a microservice scoped to a supporting technical capability adheres to the second microservice characteristic.

REPLACEABLE AND MAINTAINABLE BY A SMALL TEAM

Microservices scoped to a supporting technical capability tend to be narrowly and clearly scoped. On the other hand, part of the point of implementing such capabilities in separate microservices is that they can be complex. In other words, microservices scoped to a supporting technical capability tend to be small, which points toward adhering to the microservice characteristics of replaceability and maintainability; but the code inside them may be complex, which makes them harder to maintain and replace.

This is an area where there's a certain back and forth between using supporting technical capabilities to scope microservices on one hand, and the characteristics of microservices on the other. If a supporting technical microservice is becoming so complex that it will be hard to replace, this is a sign that you should probably look closely at the capability and try to find a way to break it down further. As in the example about notification (see section 4.2.2).

4.5.3 Tertiarily scoping to support efficiency of work

The third driver for scoping microservices is supporting efficiency of work. Let's see how making sure teams can work efficiently align with the microservice characteristics.

INDIVIDUALLY DEPLOYABLE

Part of making sure that teams can work efficiently and autonomously is to make sure that they can deliver their work all the way to production. This aligns nicely to the second microservice characteristic. When a microservices is individually deployable the team can deploy it to production without coordinating with other teams. This is part of making sure that the team can work efficiently.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

A SMALL TEAM CAN MAINTAIN A FEW HANDFULS OF MICROSERVICES

Teams should have knowledge necessary to maintain the microservices they are responsible for. Otherwise they cannot work efficiently and autonomously. When they have the knowledge needed they can maintain their microservices by themselves end-to-end which supports the fifth microservices characteristic.

4.6 Summary

- The primary driver in scoping microservices is identifying business capabilities. Business capabilities are the things an organization does that contribute to fulfilling business goals.
- You can use techniques from domain-driven design to identify business capabilities. Domain-driven design is a powerful tool for gaining better and deeper understanding of a domain. That kind of understanding enables you to identify business capabilities.
- The secondary driver in scoping microservices is identifying supporting technical capabilities. A supporting technical capability is a technical function needed by one or more microservices scoped to business capabilities.
- Supporting technical capabilities should be moved to their own microservices only if they’re sufficiently complex to be a problem in the microservices they would otherwise be part of, and if they can be individually deployed.
- Identifying supporting technical capabilities is an opportunistic form of design. You should only pull a supporting technical capability into a separate microservice if it will be an overall simplification.
- The tertiary driver in scoping microservices is efficiency of work. A team assigned to develop and maintain a microservices should be able to work efficiently and autonomuosly.
- When you’re in doubt about the scope of a microservice, lean toward making the scope slightly bigger rather than slightly smaller.
- Because scoping microservices well is difficult, you’ll probably be in doubt sometimes. You’re also likely to get some of the scopes wrong in your first iteration.
- You must expect to have to carve new microservices out of existing ones from time to time.
- You can use your doubt about scope to organize the code in your microservices so that they lend themselves to carving out new microservices at a later stage.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

5 Microservice collaboration

This chapter covers

- Understanding how microservices collaborate through commands, queries, and events
- Comparing event-based collaboration with collaboration based on commands and queries
- Implementing an event feed
- Implementing command-, query-, and event-based collaboration
- Deploying collaborating microservices to Kubernetes

Each microservice implements a single capability; but to deliver end user functionality, microservices need to collaborate. Microservices can use three main communication styles for collaboration: *commands*, *queries*, and *events*. Each style has its strengths and weaknesses, and understanding the trade-offs between them allows you to pick the appropriate one for each microservice collaboration. When you get the collaboration style right, you can implement loosely coupled microservices with clear boundaries.

In this chapter, I'll show you how to implement all three collaboration styles in code using HTTP GET, POST, PUT, and DELETE endpoints. I will also show how to deploy several collaborating microsservices to Kubernetes and have them collaborate as intended using all three collaboration style.

5.1 Types of collaboration: commands, queries, and events

Microservices are fine grained and narrowly scoped. To deliver functionality to an end user, microservices need to collaborate.

As an example, consider the Loyalty Program microservice from the point-of-sale system in

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

chapter 4. The Loyalty Program microservice is responsible for the Loyalty Program business capability. The program is simple: customers can register as users with the loyalty program; once registered, they receive notifications about new special offers and earn loyalty points when they purchase something. Still, the Loyalty Program business capability depends on other business capabilities, and other business capabilities depend on it. As illustrated in figure 5.1, the Loyalty Program microservice needs to collaborate with a number of other microservices.

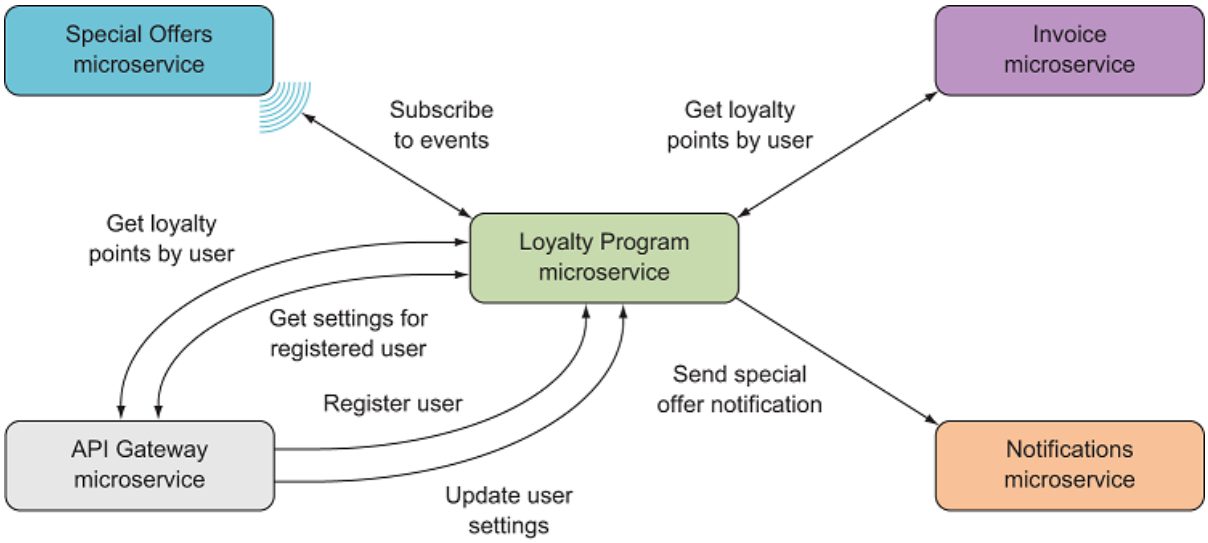

Figure 5.1 The Loyalty Program microservice collaborates with several other microservices. In some cases, the Loyalty Program microservice receives requests from other microservices; at other times, it sends requests to other microservices.

As stated in the list of microservice characteristics in chapter 1, a microservice is responsible for a single capability; and as discussed in chapter 4, that single capability is typically a business capability. End user functionalities—or use cases—often involve several business capabilities, so the microservices implementing these capabilities must collaborate to deliver functionality to the end user.

When two microservices collaborate, there are three main styles:

- *Commands*—Commands are used when one microservice needs another microservice to perform an action. For example, the Loyalty Program microservice sends a command to the Notifications microservice when it needs a notification to be sent to a registered user.
- *Queries*—Queries are used when one microservice needs information from another microservice. Because customers with many loyalty points receive a discount, the Invoice microservice queries the Loyalty Program microservice for the number of loyalty points a user has.
- *Events*—Events are used when a microservice needs to react to something that happened in another microservice. The Loyalty Program microservice subscribes to events from the Special Offers microservice so that when a new special offer is made available, it can have notifications sent to registered users.

The collaboration between two microservices can use one, two, or all three of these collaboration

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

styles. Each time two microservices need to collaborate, you must decide which style to use. Figure 5.2 shows the collaborations of Loyalty Program again, but this time identifying the collaboration style I chose for each one.

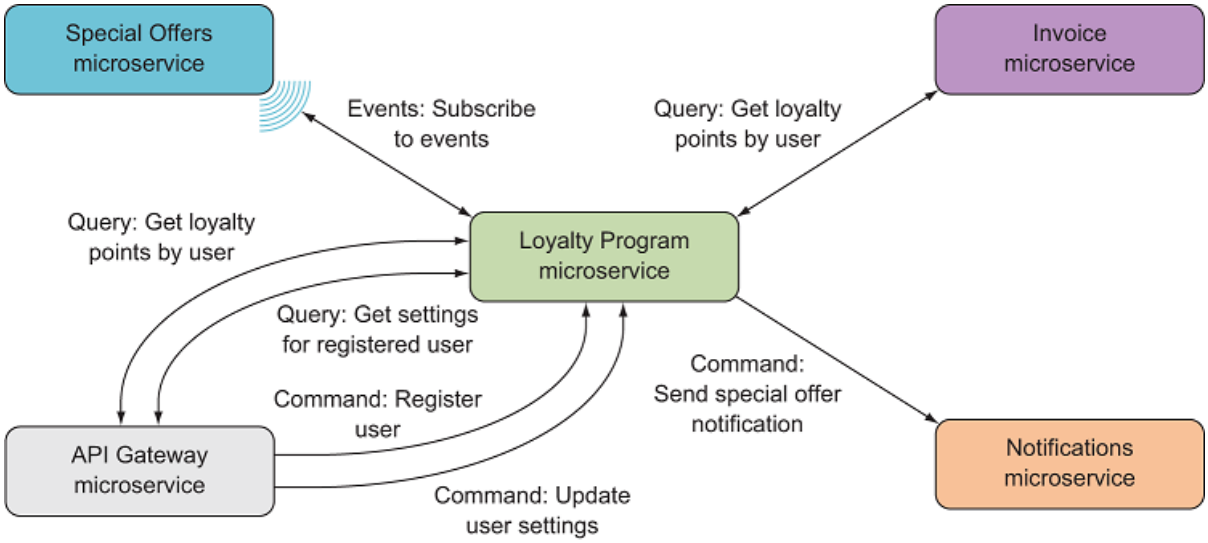

Figure 5.2 The Loyalty Program microservice uses all three collaboration styles: commands, queries, and events.

Collaboration based on commands and queries should use relatively coarse-grained commands and queries. The calls made between microservices are remote calls, meaning they cross at least a process boundary and usually also a network. This means calls between microservices are relatively slow. Even though the microservices are fine grained, you must not fall into the trap of thinking of calls from one microservice to another as being like function calls in a microservice.

Furthermore, you should prefer collaboration based on events over collaboration based on commands or queries. Event-based collaboration is more loosely coupled than the other two forms of collaboration because events are handled asynchronously. That means two microservices collaborating through events aren't temporally coupled: the handling of an event doesn't have to happen immediately after the event is raised. Rather, handling can happen when the subscriber is ready to do so. In contrast, commands and queries are synchronous and therefore need to be handled immediately after they're sent.

5.1.1 Commands and queries: synchronous collaboration

Commands and queries are both synchronous forms of collaboration. Both are implemented as HTTP requests from one microservice to another. Queries are implemented with HTTP `GET` requests, whereas commands are implemented with HTTP `POST` or `PUT` requests.

The Loyalty Program microservice can answer queries about registered users and can handle commands to create or update registered users. Figure 5.3 shows the command- and query-based collaborations that Loyalty Program takes part in.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

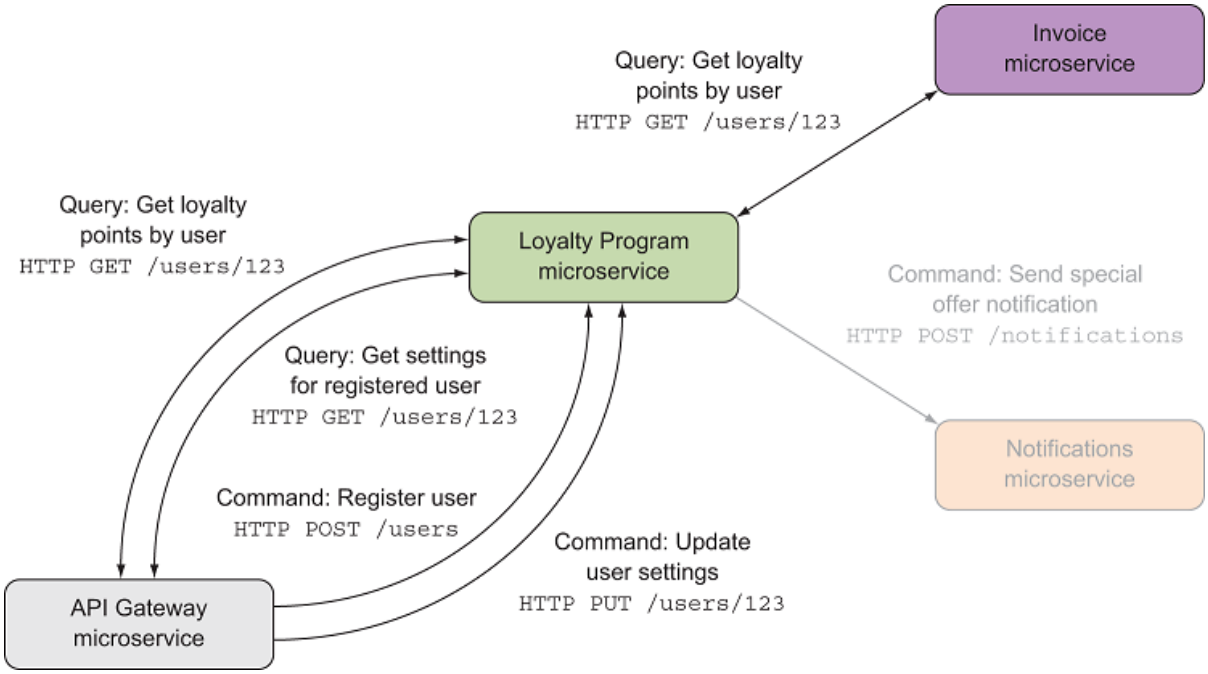

Figure 5.3 The Loyalty Program microservice collaborates with three other microservices using commands and queries. The queries are implemented as HTTP GET requests, and the commands are implemented as HTTP POST or PUT requests. The command collaboration with the Notifications microservice is grayed out because I'm not going to show its implementation—it's done exactly the same way as the other collaborations.

Figure 5.3 includes two different queries: "Get loyalty points for registered user" and "Get settings for registered user." You'll handle both of these with the same endpoint that returns a representation of the registered user. The representation includes both the number of loyalty points and the settings. You do this for two reasons: it's simpler than having two endpoints, and it's also cleaner because the Loyalty Program microservice gets to expose just one representation of the registered user instead of having to come up with specialized formats for specialized queries.

Two commands are sent to Loyalty Program in figure 5.3: one to register a new user, and one to update an existing registered user. You'll implement the first with an HTTP `POST` and the second with an HTTP `PUT`. This is standard usage of `POST` and `PUT` HTTP methods. `POST` is often used to create a new resource, and `PUT` is defined in the HTTP specification to update a resource.

All in all, the Loyalty Program microservice needs to expose three endpoints:

- An HTTP `GET` endpoint at URLs of the form /users/{userId} that responds with a representation of the user. This endpoint implements both queries in figure 5.3
- An HTTP `POST` endpoint at /users/ that expects a representation of a user in the body of the request and then registers that user in the loyalty program.
- An HTTP `PUT` endpoint at URLs of the form /users/{userId} that expects a representation of a user in the body of the request and then updates an already-registered user.

The Loyalty Program microservice is made up of the same set of standard components you've

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

seen before, as shown in figure 5.4. The endpoints are implemented in the HTTP API component.

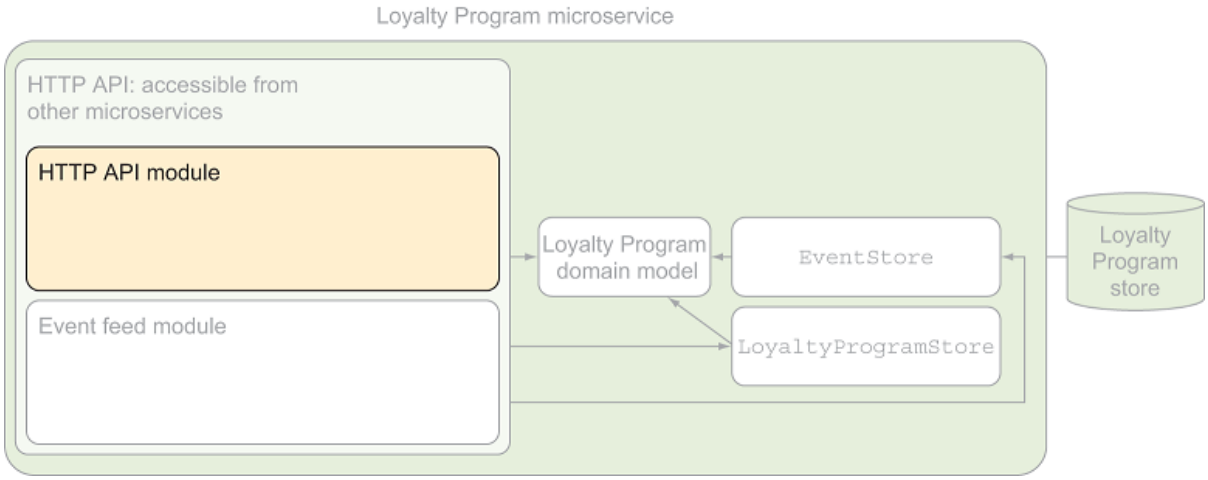

Figure 5.4 The endpoints exposed by the Loyalty Program microservice are implemented in the HTTP API component.

The other sides of these collaborations are microservices that most likely follow the same standard structure, with the addition of a `LoyaltyProgramClient` component. For instance, the Invoice microservice might be structured as shown in figure 5.5.

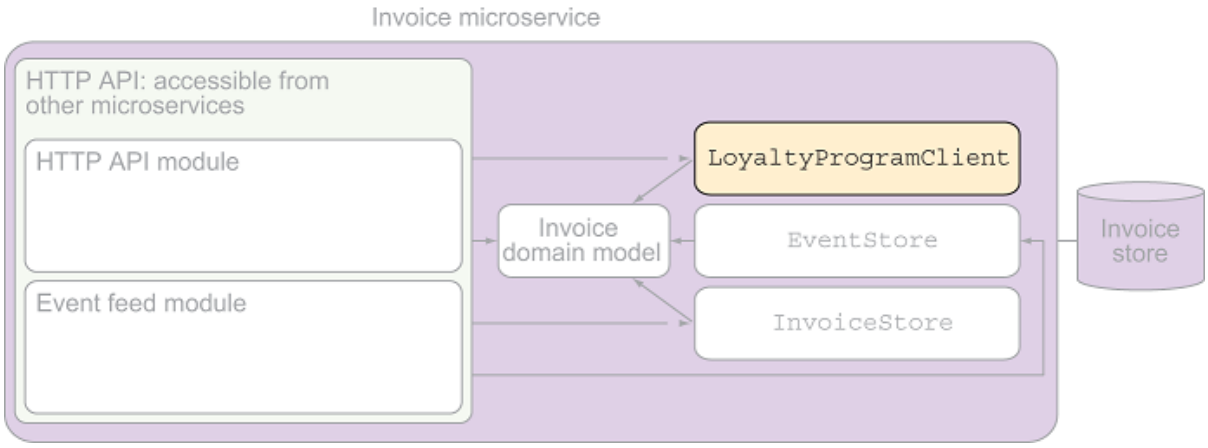

Figure 5.5 The Invoice microservice has a LoyaltyProgramClient component responsible for calling the Loyalty Program microservice.

The representation of a registered user that the Loyalty Program will expect to receive in the commands and with which it will respond to queries with is a serialization of the following `LoyaltyProgramUser`:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.1 The Loyalty Program microservice's user representation

```
 language="sql" linenumbering="unnumbered">public class LoyaltyProgramUser
{
  public int Id { get; set; }
  public string Name { get; set;  }
  public int LoyaltyPoints { get; set;  }
  public LoyaltyProgramSettings Settings { get; set; }
}

public class LoyaltyProgramSettings
{
  public string[] Interests { get; set; }
}
```

The definitions of the endpoints and the two classes in this code effectively form the contract that the Loyalty Program microservice publishes. The `LoyaltyProgramClient` component in the Invoice microservice adheres to this contract when it makes calls to the Loyalty Program microservice, as illustrated in figure 5.6.

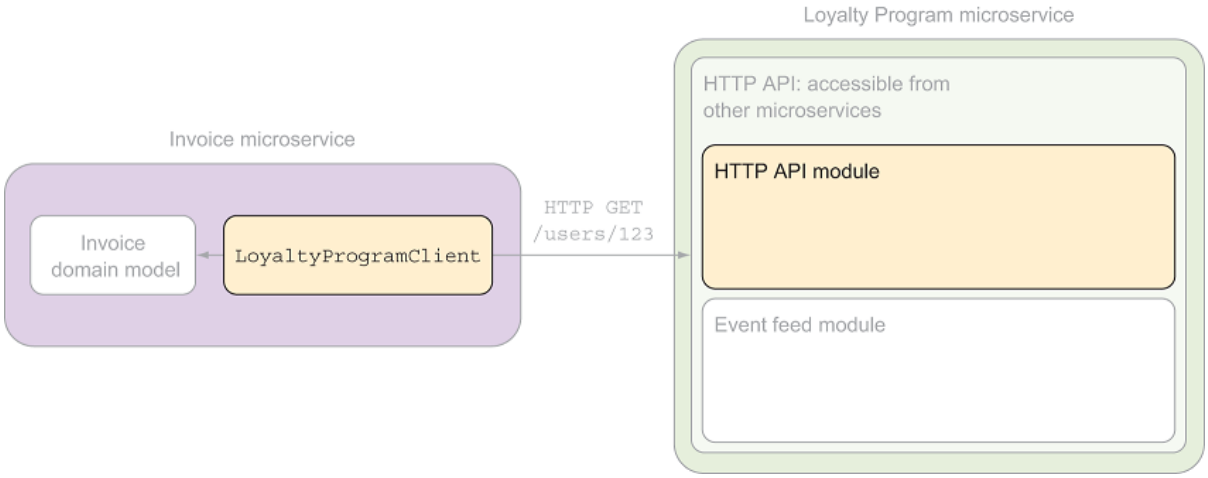

Figure 5.6 The LoyaltyProgramClient component in the Invoice microservice is responsible for making calls to the Loyalty Program microservice. It translates between the contract published by Loyalty Program and the domain model of Invoice.

Commands and queries are powerful forms of collaboration, but they both suffer from being synchronous by nature. As mentioned earlier, that creates coupling between the microservices that expose the endpoints and the microservices that call the endpoints. Next, we'll turn our attention to asynchronous collaboration through events.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

5.1.2 Events: asynchronous collaboration

Collaboration based on events is asynchronous. That is, the microservice that publishes the events doesn't call the microservices that subscribe to the events. Rather, the subscribers process new events when they're ready to process them. When we use HTTP to implement this style of collaboration the subscribes poll for new events. That polling is what I'll call *subscribing* to an event feed. Although the polling is made out of synchronous requests, the collaboration is asynchronous because publishing events is independent of any subscriber polling for events.

In figure 5.7, you can see the Loyalty Program microservice subscribing to events from the Special Offers microservice. Special Offers can publish events whenever something happens in its domain, such as every time a new special offer becomes active. Publishing an event, in this context, means storing the event in Special Offers. Loyalty Program won't see the event until it makes a call to the event feed on Special Offers. When that happens is entirely up to Loyalty Program. It can happen right after the event is published or at any later point in time.

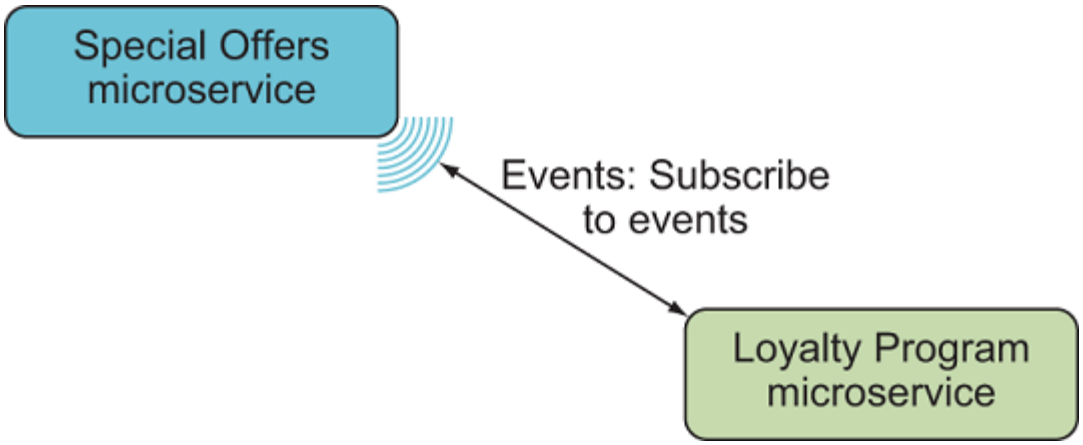

Figure 5.7 The Loyalty Program microservice processes events from the Special Offers microservice when it's convenient for Loyalty Program.

As with the other types of collaboration, there are two sides to event-based collaboration. One side is the microservice that publishes events through an event feed, and the other is the micro-services that subscribe to those events.

EXPOSING AN EVENT FEED

A microservice can publish events to other microservices via an *event feed*, which is just an HTTP endpoint—at /events, for instance—to which that other microservice can make requests and from which it can get event data. Figure 5.8 shows the components in the Special Offers microservice. Once again, the microservice has the same standard set of components that you've seen several times already. In figure 5.8, the components involved in implementing the event feed are highlighted.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

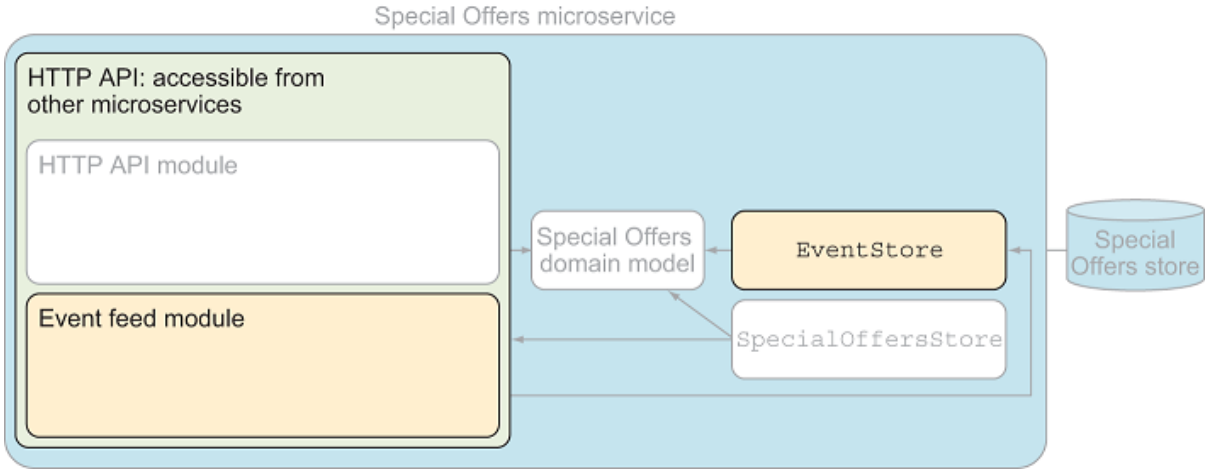

Figure 5.8 The event feed in the Special Offers microservice is exposed to other microservices over HTTP and is based on the event store.

The events published by the Special Offers microservice are stored in its database. The `EventStore` component has the code that reads events from that database and that writes them to the database. The domain model code can use `EventStore` to store the events it needs to publish. The `Event Feed` component is the implementation of the HTTP endpoint that exposes the events to other microservices: that is, the /events endpoint.

The `Event Feed` component uses `EventStore` to read events from the database and then returns the events in the body of an HTTP response. Subscribers can use query parameters to control which and how many events are returned.

SUBSCRIBING TO EVENTS

Subscribing to an event feed essentially means you poll the events endpoint of the microservice that you subscribe to. At intervals, you send an HTTP `GET` request to the /events endpoint to check whether there are any events you haven't processed yet.

Figure 5.9 is an overview of the Loyalty Program microservice, which shows that it consists of two processes. We've already talked about the web process, but the event-subscriber process is new.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

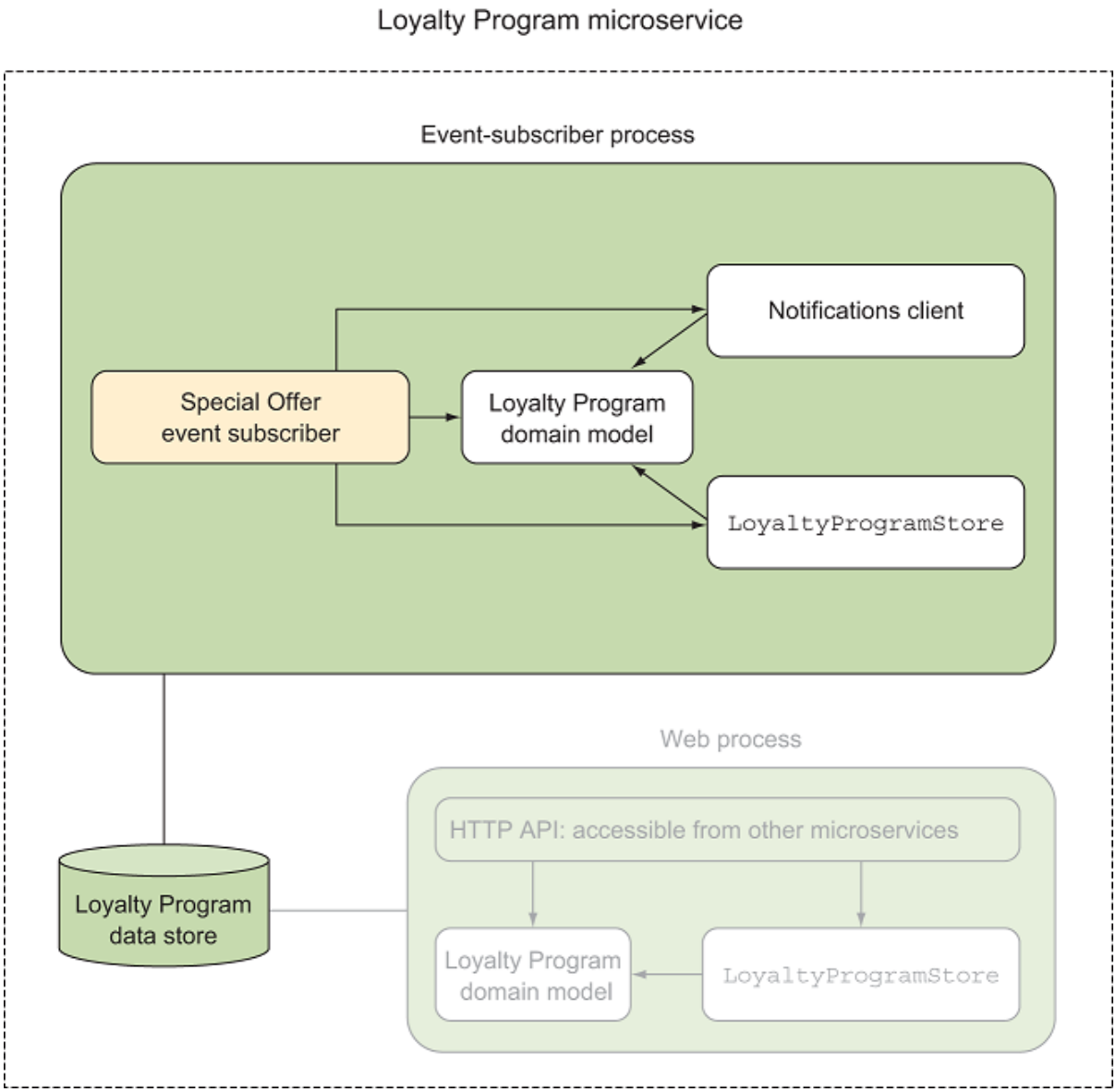

Figure 5.9 The event subscription in the Loyalty Program microservice is handled in a event-subscriber process.

The event-subscriber process is a background process that periodically makes requests to the event feed on the Special Offers microservice to get new events. When it gets back new events, it processes them by sending commands to the Notifications microservice to notify registered users about new special offers. The `SpecialOffersSubscriber` component is where the polling of the event feed is implemented, and the `Notifications-Client` component is responsible for sending the command to Notifications. The rate a which the event subscribes gets events can differ quite a bit based on the specific use case. Often events are fetched every 30 seconds or every few mintues, but there are also cases where fetching events very often, like every second, or where fetching events very seldom, like every hour, is the right thing to do.

This is the way you implement event subscriptions: microservices that need to subscribe to events have a subscriber process with a component that polls the event feed. When new events

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

are returned from the event feed, the subscriber process handles the events based on business rules.

SIDEBAR **Events over queues**

An alternative to publishing events over an event feed is to use a queue technology, like RabbitMQ or, AWS SQS, or Azure Queues. In this approach, microservices that publish events push them to a queue, and subscribers read them from the queue. Events must be routed from the publisher to multiple subscribers, and how that's done depends on the choice of queue technology and may require additional technologies on top of the queue. As with the event-feed approach, the microservice subscribing to events has an event-subscriber process that reads events from the queue and processes them. Event replay can also be built with queues, but may require a bit of extra work depending on the queue technology.

This is a viable approach to implementing event-based collaboration between microservices. But this book uses HTTP-based event feeds for event-based collaboration because it allows events to be replayed at any and it's a simple yet robust and scalable solution.

5.1.3 Data formats

So far, we've focused on exchanging data in JSON format. JSON works well in many situations, but there are reasons you might want something else:

- If you need to exchange a lot of data, a more compact format may be needed. Text-based formats such as JSON and XML are a lot more verbose than binary formats like protocol buffers.
- If you need a more structured format than JSON that's still human readable, you may want to use YAML.
- If your company uses proprietary data formatting, you may need to support that format.

In all these cases, you need endpoints capable of receiving data in another format than JSON, and they also need to be able to respond in that other format. As an example, a request to register a user with the Loyalty Program microservice using YAML in the request body looks like this:

```
POST /users HTTP/1.1 Host: localhost:5000 Accept: application/yaml Content-Type: application/
Name: Christian Settings: Interests: - whisky - cycling - "software design"
```

The response to this request also uses YAML:

```
HTTP/1.1 201 Created Content-Type: application/yaml Location: http://localhost:5000/users/1
Id: 1 Name: Christian Settings: Interests: - whisky - cycling - "software design"
```

Both the preceding request and response have YAML-formatted bodies, and both specify that the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

body is YAML in the `Content-Type` header. The request uses the `Accept` header to ask for the response in YAML. This example shows how microservices can communicate using different data formats and how they can use HTTP headers to tell which formats are used.

5.2 Implementing collaboration

This section will show you how to code the collaborations you saw earlier in figure 5.2. I'll use the Loyalty Program microservice as a starting point, but I'll also go into some of its collaborators—the API Gateway microservice, the Invoice microservice, and the Special Offers microservice—in order to show both ends of the collaborations.

Three steps are involved in implementing the collaboration:

1. Set up a project for Loyalty Program. Just as you've done before, you'll create an empty ASP.NET Core application and add MVC Core.
2. Implement the command- and query-based collaborations shown in figure 5.2. You'll implement all the commands and queries that Loyalty Program can handle, as well as the code in collaborating microservices that use them.
3. Implement the event-based collaboration shown in figure 5.2. You'll start with the event feed in Special Offers and then move on to implement the subscription in Loyalty Program. the subscription will be in a new project in Loyalty Program and will be controlled by a Kubernetes CronJob. After these steps, you'll have implemented all the collaborations of Loyalty Program.

The Loyalty Program microservice consists of a web process that has the same structure you've seen before. This is illustrated at the bottom of figure 5.10. Later, when you implement the event-based collaboration, you'll add another component that I call the *event-subscriber*. This is shown at the top of figure 5.10.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

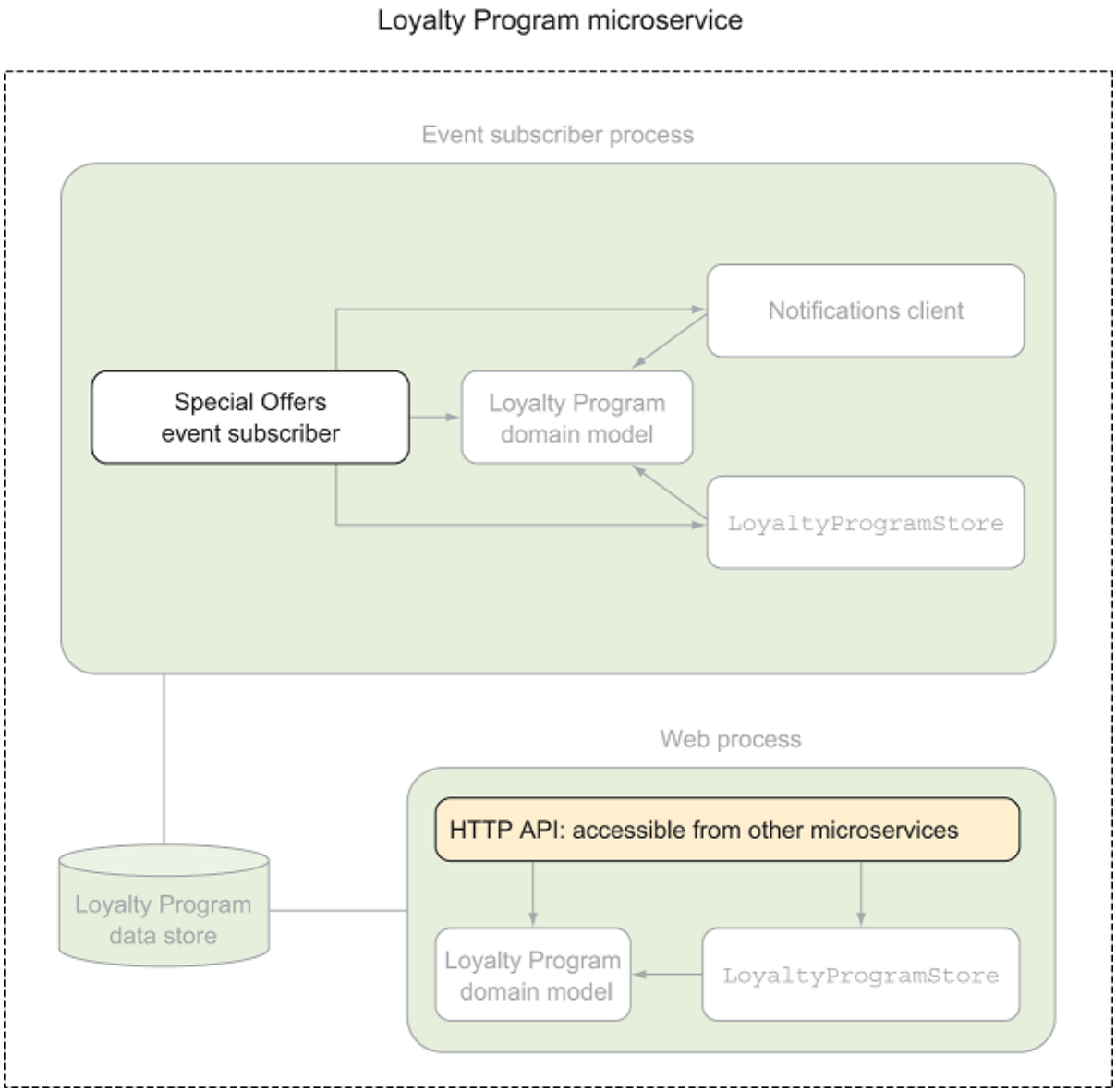

Figure 5.10 The Loyalty Program microservice has a web process that follows the structure you've seen before and an event-subscriber that handles the subscription to events from the Special Offers microservice. I'll only show the code for the highlighted components in this chapter.

In the interest of focusing on the collaboration, I won't show all the code in the Loyalty Program microservice. Rather, I'll include the code for the HTTP API in the web process, and the special offer event subscriber in the event-subscriber process.

5.2.1 Setting up a project for Loyalty Program

The first thing to do in implementing the Loyalty Program microservice is to create an empty ASP.NET Core application and add MVC Core to it. You've already done this a couple of times—in chapters 1 and 2—so I won't go over the details again here, but only remind you that your `startup.cs` should look like this:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.2 Startup class with MVC

```
 language="sql" linenumbering="unnumbered">namespace LoyaltyProgram
{
  using Microsoft.AspNetCore.Builder;
  using Microsoft.AspNetCore.Hosting;
  using Microsoft.Extensions.DependencyInjection;

  public class Startup
  {
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
      app.UseHttpsRedirection();
      app.UseRouting();
      app.UseEndpoints(endpoints => endpoints.MapControllers());
    }
  }
}
```

5.2.2 Implementing commands and queries

You now have a web project ready to host the implementations of the endpoints exposed by the Loyalty Program microservice. As listed earlier, these are the endpoints:

- An HTTP `GET` endpoint at URLs of the form /users/{userId} that responds with a representation of the user. This endpoint implements both queries in figure 5.3.
- An HTTP `POST` endpoint at /users/ that expects a representation of a user in the body of the request and then registers that user in the loyalty program
- An HTTP `PUT` endpoint at URLs of the form /users/{userId} that expects a representation of a user in the body of the request and then updates an already-registered user.

You'll implement the command endpoints first and then the query endpoint.

5.2.3 Implementing commands with HTTP POST or PUT

The code needed in the Loyalty Program microservice to implement the handling of the two commands—the HTTP `POST` to register a new user and the HTTP `PUT` to update one—is similar to the code you saw in chapter 2. You'll start by implementing an action method for the command to register a user. A request to Loyalty Program to register a new user is shown in the following listing.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.3 Request to register a user named Christian

```
 language="sql" linenumbering="unnumbered">POST /users HTTP/1.1
Host: localhost:5001
Content-Type: application/json
Accept: application/json

{                                    ❶
  "name":"Christian",
  "loyaltyPoints":0,
  "settings":{ "interests" : ["whisky", "cycling"] }
}
```

❶ JSON representation of the user being registered

To handle the command for registering a new user, you need to add an MVC controller to Loyalty Program by adding a file called UserController.cs and putting the following code in it.

Listing 5.4 POST endpoint for registering users

```
language="sql" linenumbering="unnumbered">  using System;
 using System.Collections.Generic;
 using Microsoft.AspNetCore.Mvc;

 [Route("/users")]
 public class UsersController : Controller
 {
   [HttpPost("")]
   public ActionResult<LoyaltyProgramUser> CreateUser(
     [FromBody] LoyaltyProgramUser user)                              ❶
   {
     if (user == null)
       return BadRequest();
     var newUser = RegisterUser(user);
     return Created(                                                  ❷
       new Uri($"/users/{newUser.Id}", UriKind.Relative),             ❸
       newUser);                                                      ❹
   }

   private LoyaltyProgramUser RegisterUser(LoyaltyProgramUser user)
   {
     // store the new user to a data store
   }
 }
```

❶ The request must include a LoyaltyProgramUser in the body.

❷ Uses the 201 Created status code for the response

❸ Adds a location header to the response because this is expected by HTTP for 201 Created responses

❹ Returns the user in the response for convenience

The response to the preceding request looks like this:

HTTP/1.1 201 Created Content-Type: application/json; charset=utf-8 Location: /users/4
 "name": "Christian", "loyaltyPoints": 0, "settings": { "interests": ["whisky", "cycling"] } }

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

The is not much new to notice here, except that we are using the method `Created` to control the HTTP response. There are many such convenience methods on the `Controller` base class, and throughout the book we will be using them to easily create the HTTP responses we want.

With the action method for the register-user command in place, let's turn our attention to implementing an action method for the update-user command by adding this to the `UserController`.

Listing 5.5 `PUT` endpoint for registering users

```
language="sql" linenumbering="unnumbered">    [HttpPut("{userId:int}")]
   public LoyaltyProgramUser UpdateUser(
     int userId,
     [FromBody] LoyaltyProgramUser user)
     => RegisteredUsers[userId] = user;       ❶
```

❶ Returns the new user and lets ASP.NET Core turn it into an HTTP response.

There's nothing in this code you haven't seen before.

The actions for the commands are only one side of the collaboration. The other side is the code that sends the commands. Figure 5.2 shows that the API Gateway microservice sends commands to the Loyalty Program microservice. You won't build a complete API Gateway microservice here, but in the code download for this chapter, you'll find a console application that acts as API Gateway with regard to collaborating with Loyalty Program. Here, we'll focus only on the code that sends the commands.

In the API Gateway microservice, you'll create a class called `LoyaltyProgramClient` that's responsible for dealing with communication with the Loyalty Program microservice. That class encapsulates building and sending HTTP requests to the Loyalty Program microservice.

The code for sending the register-user command takes a user name as input, creates an HTTP `POST` with a user object in the body, and sends that to the Loyalty Program microservice. After the response comes back from the Loyalty Program the we will just return the whole thing as an `HttpResponseMessage`. That's an easy solution for the current needs, but often a client class like this would also check the status code of the response, deserialize the body if the status code is as expected and possibly even perform error handling if the status code indicates an error. The following listing shows the implementation.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.6 The API Gateway microservice registering new users

```
language="sql" linenumbering="unnumbered">  using System.Net.Http;
 using System.Text;
 using System.Threading.Tasks;
 using Newtonsoft.Json;

 public class LoyaltyProgramClient
 {
   private readonly HttpClient httpClient;

   public LoyaltyProgramClient(HttpClient httpClient)
   {
     this.httpClient = httpClient;
   }

   public async Task<HttpResponseMessage> RegisterUser(string name)
   {
     var user = new {name, Settings = new { }};
     return await this.httpClient.PostAsync("/users/",       ❶
       CreateBody(user));
   }

   private static StringContent CreateBody(object user)
   {
     return new StringContent(                               ❷
       JsonConvert.SerializeObject(user),
       Encoding.UTF8,
       "application/json");                                  ❸
   }
 }
```

❶ Sends the command to Loyalty Program

❷ Serializes user as JSON

❸ Sets the Content-Type header

Similarly, `LoyaltyProgramClient` has a method for sending the update-user command. This method also encapsulates the HTTP communication involved in sending the command.

Listing 5.7 The API Gateway microservice updating users

```
language="sql" linenumbering="unnumbered">    public async Task<HttpResponseMessage> UpdateUser(dynamic
     await this.httpClient.PutAsync(                         ❶
       $"/users/{user.id}",
        CreateBody(user));
```

❶ Sends the update-user command as a PUT request

This code is similar to the code for the register-user command, except this HTTP request uses the `PUT` method. With the command handlers implemented in the Loyalty Program microservice and a `LoyaltyProgramClient` implemented in the API Gateway microservice, the command-based collaboration is implemented. API Gateway can register and update users, but it can't yet query users.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

5.2.4 Implementing queries with HTTP GET

The Loyalty Program microservice can handle the commands it needs to handle, but it can't answer queries about registered users. Remember that Loyalty Program only needs one endpoint to handle queries. As mentioned previously, the endpoint handling queries is an HTTP `GET` endpoint at URLs of the form /users/{userId}, and it responds with a representation of the user. This endpoint implements both queries in figure 5.3.

Listing 5.8 `GET` endpoint to query a user and his/her loyalty point by user ID

```
 language="sql" linenumbering="unnumbered">namespace LoyaltyProgram.Users
{
  using System;
  using System.Collections.Generic;
  using Microsoft.AspNetCore.Mvc;

  [Route("/users")]
  public class UsersController : Controller
  {
    private static readonly IDictionary<int, LoyaltyProgramUser> RegisteredUsers = new Dictionary<int, L

    [HttpGet("{userId:int}")]
    public ActionResult<LoyaltyProgramUser> GetUser(int userId) =>
      RegisteredUsers.ContainsKey(userId)
        ? (ActionResult<LoyaltyProgramUser>) Ok(RegisteredUsers[userId])
        : NotFound();

    ...
  }
}
```

There's nothing about this code that you haven't already seen several times. Likewise, the code needed in the API Gateway microservice to query this endpoint shouldn't come as a surprise:

public class LoyaltyProgramClient { ... public async Task<HttpResponseMessage> QueryUser(string this.httpClient.GetAsync($"/users/{int.Parse(arg)}"); }

This is all that's needed for the query-based collaboration. You've now implemented the command- and query-based collaborations of the Loyalty Program microservice.

5.2.5 Implementing an event-based collaboration

Now that you know how to implement command- and query-based collaborations between microservices, it's time to turn our attention to the event-based collaboration. Figure 5.11 repeats the collaborations that the Loyalty Program microservice is involved in. Loyalty Program subscribes to events from Special Offers, and it uses the events to decide when to notify registered users about new special offers.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

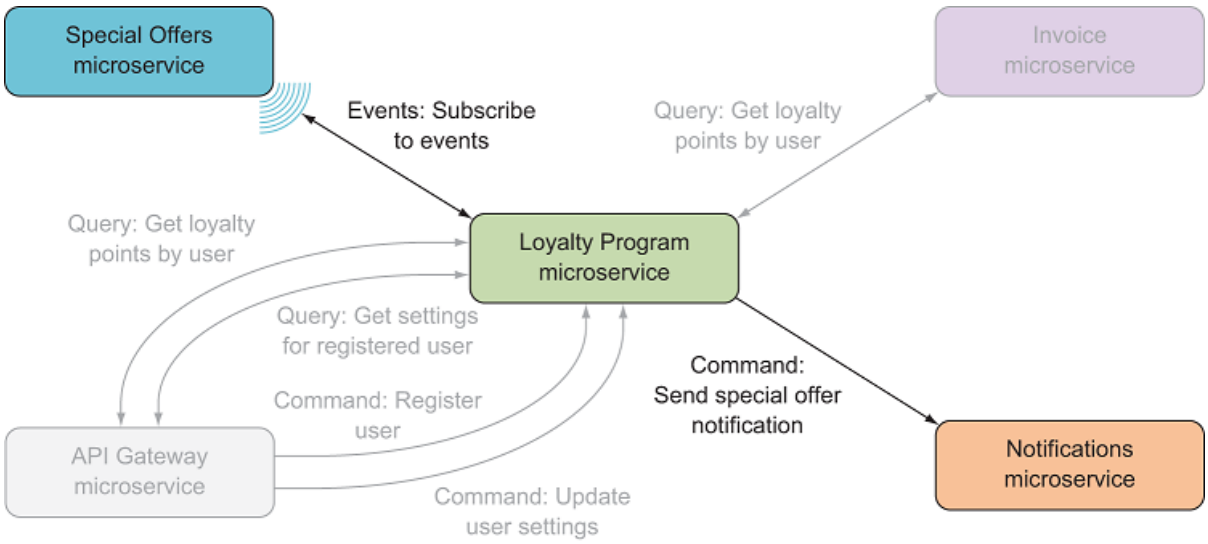

Figure 5.11 The event-based collaboration in the Loyalty Program microservice is the subscription to the event feed in the Special Offers microservice.

We'll first look at how Special Offers exposes its events in a feed. Then, we'll return to Loyalty Program and add a second process to that service, which will be responsible for subscribing to events and handling events.

IMPLEMENTING AN EVENT FEED

You saw a simple event feed in chapter 2. The Special Offers microservice implements its event feed the same way: it exposes an endpoint—`/events`—that returns a list of sequentially numbered events. The endpoint can take two query parameters—`start` and `end`—that specify a range of events. For example, a request to the event feed can look like this:
GET /events?start=10&end=110 HTTP/1.1 Host: localhost:5002 Accept: application/json

The response to this request might be the following, except that I've cut off the response after five events:

HTTP/1.1 200 OK Content-Type: application/json; charset=utf-8 [{ "sequenceNumber": 1, "occuredA
"2020-06-16T20:13:53.6678934+00:00", "name": "SpecialOfferCreated", "content": { "description": "I
ever!!!", "id": 0 } }, { "sequenceNumber": 2, "occuredAt": "2020-06-16T20:14:22.6229836+00:00", "n
"SpecialOfferCreated", "content": { "description": "Special offer - just for you", "id": 1 } }, { "sequenc
"occuredAt": "2020-06-16T20:14:39.841415+00:00", "name": "SpecialOfferCreated", "content": { "de
"Nice deal", "id": 2 } }, { "sequenceNumber": 4, "occuredAt": "2020-06-16T20:14:47.3420926+00:00"
"SpecialOfferUpdated", "content": { "oldOffer": { "description": "Nice deal", "id": 2 }, "newOffer": { "
"Best deal ever - JUST GOT BETTER", "id": 0 } } }, { "sequenceNumber": 5, "occuredAt":
"2020-06-16T20:14:51.8986625+00:00", "name": "SpecialOfferRemoved", "content": { "offer": { "des
"Special offer - just for you", "id": 1 } } }]

Notice that the events have different names (`SpecialOfferCreated`, `SpecialOfferUpdated`, and `SpecialOfferRemoved`) and the different types of events don't have the same data fields.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

This is normal: different events carry different information. It's also something you need to be aware of when you implement the subscriber in the Loyalty Program microservice. You can't expect all events to have the exact same shape.

> **NOTE** I only include the most important parts of the Special Offers code here. The rest is available in the code download

The implementation of the /events endpoint in the Special Offers microservice is a simple controller, just like the one in chapter 2.

Listing 5.9 Endpoint that reads and returns events

```
 language="sql" linenumbering="unnumbered">namespace SpecialOffers.Events
{
  using System.Linq;
  using Microsoft.AspNetCore.Mvc;

  [Route(("/events"))]
  public class EventFeedController : Controller
  {
    private readonly IEventStore eventStore;

    public EventFeedController(IEventStore eventStore)
    {
      this.eventStore = eventStore;
    }

    [HttpGet("")]
    public ActionResult<EventFeedEvent[]> GetEvents([FromQuery] int start, [FromQuery] int end)
    {
      if (start < 0 || end < start)
        return BadRequest();

      return this.eventStore.GetEvents(start, end).ToArray();
    }
  }
}
```

This controller only uses ASP.NET Core features that we've already discussed. You may notice, however, that it returns the result of `eventStore.GetEvents`. ASP.NET Core serializes it as an array. The `EventEventFeedEvent` is a class that carries a little metadata and a `Content` field that's meant to hold the event data.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.10 `Event` class that represents events

```
language="sql" linenumbering="unnumbered">  public class EventFeedEvent
 {
   public long SequenceNumber { get; }
   public DateTimeOffset OccuredAt { get; }
   public string Name { get; }
   public object Content { get; }

   public EventFeedEvent(
     long sequenceNumber,
     DateTimeOffset occuredAt,
     string name,
     object content)
   {
     this.SequenceNumber = sequenceNumber;
     this.OccuredAt = occuredAt;
     this.Name = name;
     this.Content = content;
   }
 }
```

The `Content` property is used for event-specific data and is where the difference between a `SpecialOfferCreated` event, a `SpecialOfferUpdated` and a `SpecialOfferREmoved` event appears. Each has its own type of object in `Content`.

This is all it takes to expose an event feed. This simplicity is the great advantage of using an HTTP-based event feed to publish events. Event-based collaboration can be implemented over a queue system, but that introduces another complex piece of technology that you have to learn to use and administer in production. That complexity is warranted in some situations, but certainly not always.

CREATING AN EVENT-SUBSCRIBER PROCESS

Subscribing to an event feed essentially means you'll poll the events endpoint of the microservice you subscribe to. At intervals, you'll send an HTTP `GET` request to the /events endpoint to check whether there are any events you haven't processed yet.

We will implement this periodic polling as two main parts:

- A simple console application the reads one batch of events
- We will use a Kubernetes cron job to run the console application at intervals

Putting these two together they implement the event subscription: The cron job makes sure the console application runs at an interval and each time the console application runs it sends the HTTP GET request to check whether there are any events to process.

The first step in implementing an event-subscriber process is to create a console application with the following `dotnet` command:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

PS> dotnet new conolse -n EventConsumer

and run it with `dotnet` too:

PS> dotnet run

The application is empty, so nothing interesting happens yet, but in the next section we will make it read events.

SUBSCRIBING TO AN EVENT FEED

You now have a `EventConsumer` console application. All it has to do is read one batch of events and track where the starting point of the next batch of events is. This is done as follows:

Listing 5.11 Read a batch of events from an event feed

```
 language="sql" linenumbering="unnumbered">namespace EventConsumer
{
  using System;
  using System.Net.Http;
  using System.Net.Http.Headers;
  using System.Threading.Tasks;

  class Program
  {
    static int start = 0;

    static async Task Main(string[] args)
    {
      start = GetStartIdFromDataStore();            ❶
      var end = start + 100;
      var client = new HttpClient();
      client
        .DefaultRequestHeaders
        .Accept
        .Add(new MediaTypeWithQualityHeaderValue("application/json"));
      var resp =
        await client.GetAsync(                                          ❷
          new Uri($"http://special-offers/events?start={start}&end={end}"));
      var rawEvents = await resp.Content.ReadAsStringAsync();           ❸
      ProcessEvents(rawEvents);                                         ❹
      SaveStartIdToDataStore(start);                                    ❺
    }

    static int GetStartIdFromDataStore() { ... }
    static void SaveStartIdToDataStore(int start) { ... }
    static void ProcessEvents(string rawEvents) { ... }
  }
}
```

❶ Read the starting point of this batch from a database

❷ Send GET request to the event feed

❸ Read the response body. This a JSON array of events.

❹ Call method to process the events in this batch. ProcessEvents also updates the start variable.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

❺ Save starting point of the next batch of events

With the code above the `EventConsumer` can read a batch of events, and every time it is called it reads the next batch of events. The remaining part is to process the events:

Listing 5.12 Deserializing and then handling events

```
 language="sql" linenumbering="unnumbered">private void ProcessEvents(string content)
{
  var events = JsonConvert
    .DeserializeObject<IEnumerable<SpecialOfferEvent>>(content);
  foreach (var ev in events)
  {
    dynamic eventData = ev.Content;                              ❶
    // handle 'ev' using the eventData.
    start = Math.Max(start, ev.SequenceNumber + 1);     ❷
  }
}
```

❶ Treats the content property as a dynamic object

❷ Keeps track of the highest event number handled

There are a few things to notice here:

- This method keeps track of which events have been handled #2. This makes sure you don't request events from the feed that you've already processed.
- We treat the `Content` property on the events as `dynamic` #1. As you saw earlier, not all events carry the same data in the `Content` property, so treating it as dynamic allows you to access the properties you need on `.Content` and not care about the rest. This is a sound approach because you want to be liberal in accepting incoming data—it shouldn't cause problems if the Special Offers microservice decides to add an extra field to the event JSON. As long as the data you *need* is there, the rest can be ignored.
- The events are deserialized into the type `SpecialOfferEvent`. This is a different type than the `EventFeedEvent` type used to serialize the events in Special Offers. This is intentional and is done because the two microservices don't need to have the exact same view of the events. As long as Loyalty Program doesn't depend on data that isn't there, all is well.

The `SpecialOfferEvent` type used here is simple and contains only the fields used in Loyalty Program:

public struct SpecialOfferEvent { public long SequenceNumber { get; set; } public string Name { ge
  public object Content { get; set; } }

This concludes your implementation C# part of event subscriptions. The other part - the Kubernetes cron job - is implemented in the next section.

5.2.6 Deploying to Kubernetes

In this section we will deploy the LoyaltyProgram and SpecialOffers microservices to Kubernetes. To that end there are three steps we need to perform:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

- We will build Docker containers for both microservices. The container for SpecialOffers is similar to what we saw in chapter 3. The container for LoyaltyProgram has a twist: Based on an environment variable it can act either as an HTTP API exposing the POST, PUT and GET endpoint we implemented earlier or it can act as the event consumer we implemented in the previous section. This is explained in detail below.
- We will deploy the SpecialOffers microservice similarly to how we deployed to Kubernetes in chapter 3
- We will deploy LoyaltyProgram, which will consist of deploying the LoyaltyProgram container in two copies different configurations; one running as the API and one running as the event consumer.

> **NOTE** You can find the code for the Special Offers microservice in the code download. Here I only show the deployment bits for the Special Offers microservice

With those three steps done we will have three parts running in Kubernetes:

- A pod for the Special Offers microservice
- A pod for the API part of the Loyalty Program microservice
- A cron job for the event consumer part of the Loyalty Program.

We will also have two deployments in Kubernetes, one for the Special Offers microservice and one for the Loyalty Program microservice. Since we consider the Loyalty Program API and the Loyalty Program event consumer as parts of the same Loyalty Program microservice will always deploy these two together and by the same token we will put them into the same Kubernetes manifest file. At run time they run independently, though, which allows us to for instance scale up the API part to deal with higher load while still only running one event consumer.

5.2.7 Building a Docker container Special Offers microservice ====

Before we can deploy the Special Offers microservice to Kubernetes we need to build a Docker container for it. To do that we use this Dockerfile which is similar to the Dockerfile for the Shopping Cart we saw in chapter 2:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.13 Dockerfile for Special Offers microservice

```
 language="sql" linenumbering="unnumbered">FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src
COPY . .
RUN dotnet restore "SpecialOffers.csproj"
WORKDIR "/src"
RUN dotnet build "SpecialOffers.csproj" -c Release -o /api/build

FROM build AS publish
WORKDIR "/src"
RUN dotnet publish "SpecialOffers.csproj" -c Release -o /api/publish

FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS final
WORKDIR /app
EXPOSE 80
COPY --from=publish /api/publish ./api
ENTRYPOINT dotnet api/SpecialOffers.dll
```

This build the special offers microservice and when started will run the special offers microservice. There is nothing new here - we are just applying what we learned in chapter 2.

> **NOTE** Recall from chapter 2 that the `docker build` command builds a container from a Dockerfile

5.2.8 Building a Docker container for both parts of Loyalty Program ====

Next up we build a container image for LoyaltyProgram microservice. The Loyalty Program has two parts: The API and the event consumer. We will build one container that is capable on running either as the API or the event consumer based on an environment variable. To do that we create a Dockerfile in the LoyaltyProgram code base where the `ENTRYPOINT` is controlled by an environment variable:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.14 One docker image for the LoyaltyProgram which can run either the API of the event consumer

```
 language="sql" linenumbering="unnumbered">FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src
COPY . .
RUN dotnet restore "LoyaltyProgram/LoyaltyProgram.csproj"
RUN dotnet restore "EventConsumer/EventConsumer.csproj"
WORKDIR "/src/LoyaltyProgram"
RUN dotnet build "LoyaltyProgram.csproj" -c Release -o /api/build  ❶
WORKDIR "/src/EventConsumer"
RUN dotnet build "EventConsumer.csproj" -c Release -o /consumer/build/consumer ❷

FROM build AS publish
WORKDIR "/src/LoyaltyProgram"
RUN dotnet publish "LoyaltyProgram.csproj" -c Release -o /api/publish   ❸
WORKDIR "/src/EventConsumer"
RUN dotnet publish "EventConsumer.csproj" -c Release -o /consumer/publish  ❹

FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS final
WORKDIR /app
EXPOSE 80
COPY --from=publish /api/publish ./api
COPY --from=publish /consumer/publish ./consumer
ENTRYPOINT dotnet $STARTUPDLL                                    ❺
```

❶ Builds the API

❷ Builds the event consumer

❸ Publishes the API

❹ Publishes the event consumer

❺ Use the value of the STARTUPDLL environment variable as the entry point

This Dockerfile is a little different from what we have seen before. The Dockerfiles we have seen before have built and published one ASP.NET Core application. This one builds and publishes an ASP.NET Core application - the Loyalty Program - and a .NET Core console application - the event consumer. The image built with this Dockerfile will therefore have both the `LoyaltyProgram.dll` and the `EventConsumer.dll` in it. The last line of the Dockerfile, `ENTRYPOINT dotnet $STARTUPDLL` says that when the images built with this Dockerfile starts up they should run `dotnet` on whatever the environment variable `$STARTUPDLL` points to. So id `$STARTUPDLL` is equal to the path to the `LoyaltyProgram.dll` the image will run the Loyalty Program API and it '$STARTUPDLL' is equal to the path to the `EventConsumer.dll` the image will run the Loyalty Program event consumer.

Let’s see this Dockerfile in action. First we build a Loyalty Program Docker image:

> docker build . -t loyalty-programn

Now we run the Loyalty-Program image as either the API part of the event consumer part. Let’s first try to run it as the API by passing in the path to the `LoyaltyProgram.dll` in the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

`$STARTUPDLL`:

> docker run --rm -p 5001:80 -e STARTUPDLL="api/LoyaltyProgram.dll" loyalty-program

That command also maps port 5001 on your localhost to port 80 in the container. The Loyalty Program API is now running and you can call it on localhost port 5001. Let's also run the same Docker image as the Loyalty Program API:

> docker run --rm -e STARTUPDLL="consumer/EventConsumer.dll" loyalty-program

This will run the event consumer, but it will fail to fetch events from the Special Offers event feed, because we do not have one running. We fix that by running the Special Offer microservice in a container and putting that on the same Docker network as the event consumer. To that we create a new Docker network called microservices:

> docker network create --driver=bridge microservices

This creates a Docker network called microservices. The containers we add to the microservices network can communicate using the container names and host names. We will take advantage of that when we run the Special Offers microservice by adding it to the microservices network and giving it the name special-offers. To do just that run this command:

> docker run --rm -p 5002:80 --network=microservices --name=special-offers special-offers

Now we can rerun the event consumer, this time on the microservices network and it will succeed:

docker run --rm -e STARTUPDLL="consumer/EventConsumer.dll" --network=microservices loyalty-p

This run the event consumer once and it will consume one batch of events. Later we will set up a CRON schedule in Kubernetes that trigges the event consumer periodically.

We are now able to run all the parts in Docker on localhost: The Special Offers microservice and both parts of the Loyalty Program microservice at the same time and have them collaborate as intended. This confirms that we have successfully created Docker container images for the Loyalty Program and the Special Offers microservices. Next we will deploy those container images to our Kubernetes cluster in Azure.

5.2.9 Deploy the LoyalProgram API and the Special Offers ====

In chapter 3 we setup a Kubernetes cluster on Azure. We will continue to use that cluster throughout the book and now we will deploy the Loyalty Program microservice and the Special Offers microservice to it.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

NOTE Remember that in chapter 3 we created a script called `create-aks.ps1` for setting up the Kubernetes cluster and container registry in Azure. The script can be found in the code download. In case you did not create a Kubernetes cluster in chapter 3, I recommend you create one now.

First we add a file called `loyalty-program.yaml` to the Loyalty Programs code base and put the Kubernetes manifest for the Loyalty Program API into it. This is just like the Kubernetes manifest file we saw in chapter 3 except this one uses the Loyalty Program container image:

Listing 5.15 Kubernetes manifest for the LoyaltyProgram API

```
 language="sql" linenumbering="unnumbered">kind: Deployment
apiVersion: apps/v1
metadata:
  name: loyalty-program
spec:
  replicas: 1                                          ❶
  selector:
    matchLabels:
      app: loyalty-program
  template:
    metadata:
      labels:
        app: loyalty-program
    spec:
      containers:
        - name: loyalty-program
          image: microservicesindotnetregistry1.azurecr.io/loyalty-program:1.0.2
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
          env:
            - name: STARTUPDLL                         ❷
              value: "api/LoyaltyProgram.dll"          ❸
---
apiVersion: v1
kind: Service
metadata:
  name: loyalty-program
spec:
  type: LoadBalancer
  ports:
    - name: loyalty-program
      port: 5001
      targetPort: 80
  selector:
    app: loyalty-program
```

❶ The number of copies of the LoyaltyProgram API we want deployed

❷ Override the `STARTUPDLL` environment variable to control how the container starts up

❸ Point to the `LoyaltyProgram.dll` which is will run the LoyaltyProgram API

With this file in place we are ready deploy the Loyalty Program API to Kubernetes with this

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

command:

> kubectl apply -f loyalty-program.yaml

The Kubernetes manifest for the Special Offers microservice is similar and is also deployed using the `kubectl apply` command.

Once the Loyalty Program API and the Special Offers microservice is deploy we should se both running in Kubernetes both in the Kubernetes dashboard (which we configured in chapter 3) and in the `kubectl` command line:

> kubectl get pods NAME READY STATUS RESTARTS AGE loyalty-program-5d87df4656-9x89c 1 55s special-offers-67d6b78998-mttp6 1/1 Running 0 43s

We can also find the IP addresses and ports where the two APIs are available:

> kubectl get services NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE kubernetes Cluste <none> 443/TCP 20d loyalty-program LoadBalancer 10.0.137.255 40.127.231.56 5001:32553/TCP 2m special-offers LoadBalancer 10.0.76.165 52.142.115.22 80:32391/TCP 2m16s

I my case the Loyalty Programs API is available at http://40.127.231.56:5001 and the Special Offers API is at http://52.142.115.22/. The IP addresses will be different in your case since they are assigned dynamically be AKS (Azure Kubernetes Service).

That's all we need to deploy the two API.

5.2.10 Deploy EventConsumer ====

Remaining work is to deploy the Loyalty Programs event consumer. This is done by extending the Kubernetes manifest for the Loyalty Program and re-applying it using `kubectl`. As mentioned we will deploy the event consumer as a cron job that will be called on a schedule. To do that we the following to the `loyalty-prorgam.yaml` file created in the previous section:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 5.16 Kubernetes manifest for the LoyaltyProgram event consumer

```
 language="sql" linenumbering="unnumbered">---
apiVersion: batch/v1beta1                          ❶
kind: CronJob                                      ❷
metadata:
  name: loyalty-program-consumer
spec:
  schedule: "*/1 * * * *"                          ❸
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: loyalty-program
              image: microservicesindotnetregistry1.azurecr.io/loyalty-program:1.0.2
              imagePullPolicy: IfNotPresent
              env:
                - name: STARTUPDLL
                  value: "consumer/EventConsumer.dll"  ❹
          restartPolicy: Never
  concurrencyPolicy: Forbid                        ❺
```

❶ The Kubernetes API version needed to specify a cron job

❷ Indicate that this is a cron job

❸ Define the schedule for this job.

❹ Point to the event consumer dll

❺ Make sure only one copy at a time of the event consumers runs

The schedule defined here means the event consumer will run once every minut. This can be adjusted based on the specifics of the event consumer. Some events are rare and do not require immediate reaction. In those cases the schedule can be much slower. Some events happen often in which case we might want to keep a one minute schedule but read larger batches every time the event consumer runs.

To deploy the event consumer we use the `loyalty-program.yaml` again:

> kubectl apply -f loyalty-program.yaml

Re-running this command is quite fine. Kubernetes will figure out if there are any changes to made, and apply only the changes.

Now that the event consumer has also been deployed we should see a cron job in Kubernetes too:

> kubectl get cronjob NAME SCHEDULE SUSPEND ACTIVE LAST SCHEDULE AGE
loyalty-program-consumer */1 * * * * False 0 20s 1m20s

We can also see from this that the Loyalty Program event consumer by looking at the last shedule column. In my case the cron job has been triggered 20 seconds ago.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

We can also see a list of recent invocation of the Loyalty Program event consumer by looking at the Kubernetes jobs list:

```
> kubectl get jobs NAME COMPLETIONS DURATION AGE loyalty-program-consumer-159075594
loyalty-program-consumer-1590757080 1/1 2s 1m11s loyalty-program-consumer-1590757140 1/1 3s 2
```

This concludes the deployment of the Loyalty Program microservice. The API part as well as the event consumers part is running in our Kubernetes cluster in AKS. The Kubernetes manifest we have created for the Loyalty Program is the template for the manifests for all the microservices we will create, only the container images will have different name and the event consumer schedules will vary.

If we want to convince our selves that the event consumer is indeed running we can inspect the logs of one of the instantiation of the cron job:

```
> kubectl logs job.batch/loyalty-program-consumer-1590755940
[{"sequenceNumber":1,"occuredAt":"2020-06-18T18:07:13.7973414+00:00","name":"SpecialOfferCr
deal","id":0}},{"sequenceNumber":2,"occuredAt":"2020-06-18T18:07:17.7957514+00:00","name":"S
deal","id":1}},{"sequenceNumber":3,"occuredAt":"2020-06-18T18:07:47.4246091+00:00","name":"S
deal","id":1},"newOffer":{"description":"Best deal ever - JUST GOT
BETTER","id":0}}},{"sequenceNumber":4,"occuredAt":"2020-06-18T18:08:04.5908816+00:00","nan
deal","id":0}}}]
```

Looking at these logs we will be able to see special offers event being consumed shortly after we make calls to the Special Offers API.

With both the Special Offers microservice and the Loyalty Program microservice running in AKS we have finished implementing examples of all three styles of collaboration: Commands, queries, and events.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

5.3 Summary

- There are three types of microservice collaboration:
  - Command-based collaboration, where one microservice uses an HTTP `POST` or `PUT` to make another microservice perform an action
  - Query-based collaboration, where one microservice uses an HTTP `GET` to query the state of another microservice
  - Event-based collaboration, where one microservice exposes an event feed that other microservices can subscribe to by polling the feed for new events
- Event-based collaboration is more loosely coupled than command- and query-based collaboration.
- You can use `HttpClient` to send commands to other microservices and to query other microservices.
- You can use MVC controllers to expose the endpoints for receiving and handling commands and queries.
- An MVC controller can expose a simple event feed.
- You can create a process that subscribes to events by
  - Creating a .NET Core console application
  - Using `HttpClient` to read events from an event feed
  - Running this application as a cron job in Kubernetes

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

6 Data ownership and data storage

In this chapter:

- Which data microservices store
- Understanding how data ownership follows business
- capabilities Using data replication for speed and
- robustness
- Building read models from event feeds with event subscribers Implementing data storage in microservices

Software systems create, use, and transform data. Without the data, most software systems wouldn't be worth much, and that's true for microservice systems too. In this chapter, you'll learn where a piece of data should be stored and which microservice should be responsible for keeping it up to date. Furthermore, you'll learn how you can use data replication to make your microservice system both more robust and faster.

6.1 Each microservice has a data store

One of the characteristics of microservices identified in chapter 1 is that each microservice should own its data store. The data in that data store is solely under the control of the microservice, and it's exactly the data the microservice needs. First it's data belonging to the capability the microservice implements, but it's also supporting data, like cached data and read models created from event feeds.

The fact that each microservice owns a data store means you don't need to use the same database technology for all microservices. You can choose a database technology that's suited to the data that each microservice needs to store.

A microservice typically needs to store three types of data:

- Data belonging to the capability the microservice implements. This is data that the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

microservice is responsible for and must keep safe and up to date.

- Events raised by the microservice. During command processing, the microservice may need to raise events to inform the rest of the system about updates to the data the microservice is responsible for.
- Read models based on data in events from other microservices or occasionally on data from queries to other microservices.

These three types of data may be stored in different databases and even in different types of databases.

6.2 Partitioning data between microservices

When you're deciding where to store data in a microservice system, competing forces are at play. The two main forces are data ownership and locality:

- *Ownership of data* means being responsible for keeping the data correct, safe, and up to date.
- *Locality of data* refers to where the data a microservice needs is stored. Often, the data should be stored nearby—preferably in the microservice itself.

These two forces may be at odds, and in order to satisfy both, you'll often have to store data in several places. That's OK, but it's important that only one of those places be considered the authoritative source. Figure 6.1 illustrates that whereas one microservice stores the authoritative copy of a piece of data, other microservices can mirror that data in their own data stores.

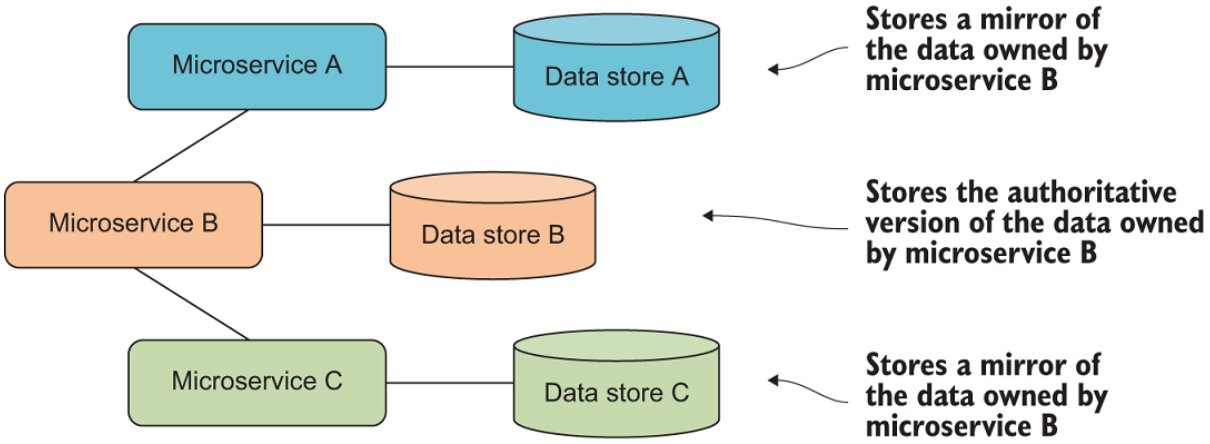

Figure 6.1 Microservices A and C collaborate with microservice B. Microservices A and C can store mirrors of the data owned by microservice B, but the authoritative copy is stored in microservice B's own data store.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

6.2.1 Rule 1: Ownership of data follows business capabilities

The first rule when deciding where a piece of data belongs in a microservices system is that ownership of data follows business capabilities. As discussed in chapter 4, the primary driver in deciding on the responsibility of a microservice is that it should handle a business capability. The business capability defines the boundaries of the microservice—everything belonging to the capability should be implemented in the microservice. This includes storing the data that falls under the business capability.

Domain-driven design teaches that some concepts can appear in several business capabilities and that the meaning of the concepts may differ slightly. Several microservices may have the concept of a customer, and they will work on and store customer entities. There may be some overlap between the data stored in different microservices, but it's important to be clear about which microservice is in charge of what.

For instance, only one microservice should own the home address of a customer. Another microservice could own the customer's purchase history, and a third the customer's notification preferences. The way to decide which microservice is responsible for a given piece of data—the customer's home address, for instance—is to figure out which business process keeps that data up to date. The microservice responsible for the business capability is responsible for storing the data and keeping it up to date.

Let's consider again the e-commerce site from chapters 1 and 2. Figure 6.2 shows an overview of how that system handles user requests for adding an item to a shopping cart. Most of the microservices in figure 6.2 are dimmed, to put the focus on three microservices: Shopping Cart, Product Catalog, and Recommendations.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

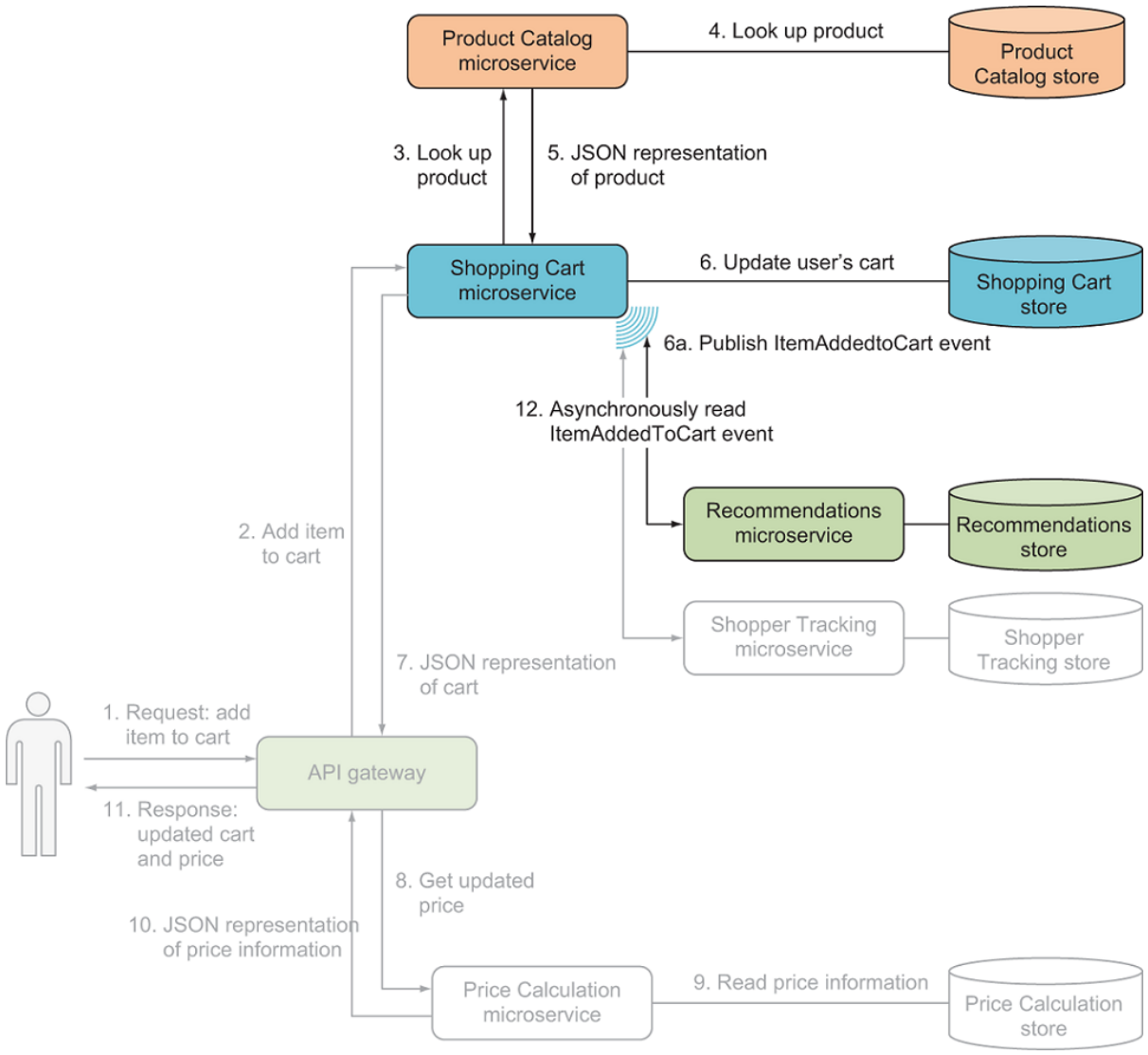

Figure 6.2 In this e-commerce example (from chapters 1 and 2), we'll focus on partitioning data between the Shopping Cart microservice, the Product Catalog microservice, and the Recommendations microservice.

Each of the highlighted microservices in figure 6.2 handles a business capability: the Shopping Cart microservice is responsible for keeping track of users' shopping carts; the Product Catalog microservice is responsible for giving the rest of the system access to information from the product catalog; and the Recommendations microservice is responsible for calculating and giving product recommendations to users of the e-commerce site. Data is associated with each of these business capabilities, and each microservice *owns* and is responsible for the data associated with its capability. Figure 6.3 shows the data each of the three microservices owns. Saying that a microservice owns a piece of data means it must store that data and be the authoritative source for that piece of data.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

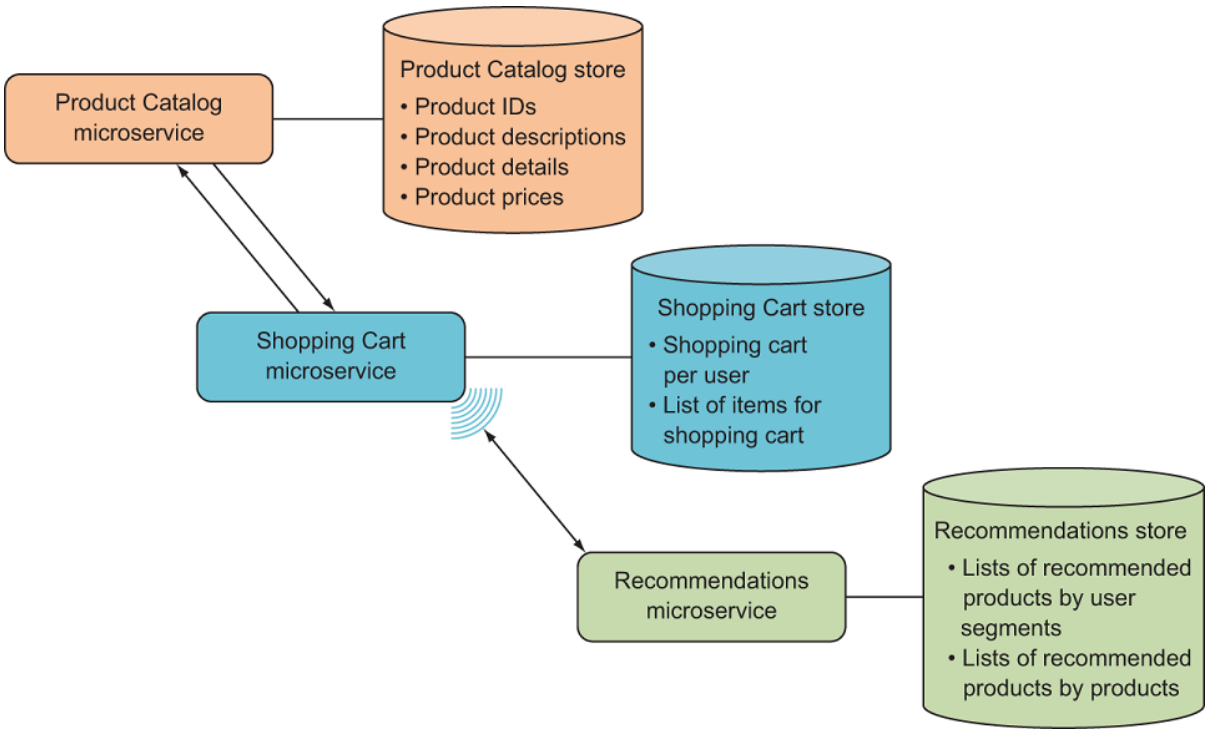

Figure 6.3 Each microservice owns the data belonging to the business capability it implements.

6.2.2 Rule 2: Replicate for speed and robustness

The second force at play when deciding where a piece of data should be stored in a microservices system is locality. There's a big difference between a microservice querying its own database for data and a microservice querying another microservice for that same data. Querying its own database is generally both faster and more reliable than querying another microservice.

Once you've decided on the ownership of data, you'll likely discover that your microservices need to ask each other for data. This type of collaboration creates a certain coupling: one microservice querying another means the first is *coupled* to the other. If the second microservice is down or slow, the first microservice will suffer.

To loosen this coupling, you can cache query responses. Sometimes you'll cache the responses as they are, but other times you can store a read model based on query responses. In both cases, you must decide when and how a cached piece of data becomes invalid. The microservice that owns the data is in the best position to decide when a piece of data is still valid and when it has become invalid. Therefore, endpoints responding to queries about data owned by the microservice should include cache headers in the response telling the caller how long it should cache the response data.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

USING HTTP CACHE HEADERS TO CONTROL CACHING

HTTP defines a number of headers that can be used to control how HTTP responses can be cached. The purpose of the HTTP caching mechanisms is twofold:

- To eliminate the need, in many cases, to request information the caller already has
- To eliminate the need, in some other situations, to send full HTTP responses

```
To eliminate the need to make requests for information the caller already has,
    the server can add a (((cache-control header)))`cache-control` header to responses.
    The HTTP specification defines a range of controls that can be set in the `cache-control`
    header. The most common are the(((private|public directive))) `private|public` and
    the(((max-age value))) `max-age` directives. The first indicates whether only the
    caller —`private` —may cache the response or if intermediaries—proxy servers,
    for instance—may cache the response, too. The `max-age` directive indicates
    the number of seconds the response may be cached. For example, the following
    `cache-control` header indicates that the caller,and only the caller, can cache
    the response for 3,600 seconds:
```

```
cache-control: private, max-age:3600
```

That is, the caller, may reuse the response any time it wants to make an HTTP request to the same URL with the same method—`GET`, `POST`, `PUT`, `DELETE`—and the same body within 3,600 seconds. In this book I will only use caching for queries, which means only for `GET` requests. It's worth noting that the query string is part of the URL, so caching takes query strings into account.

To eliminate the need to send a full response in cases where the caller has a cached but stale response, the `ETag` and `If-None-Match` headers can be used: The server can add an `ETag` header to responses. This is an identifier for the response. When the caller makes a later request to the same URL using the same method and the same body, it can include the `ETag` in a request header called `If-None-Match`. The server can read the `ETag` and, know which response the caller has cached. If the server decides the cached response is still valid, it can return a response with the 304 Not Modified status code to tell the client to use the already-cached response. Furthermore, the server can add a `cache-control` header to the 304 response to prolong the period the response may be cached. Note that the `ETag` is set by the server and later read again by the same server.

Let's consider the microservices in figure 6.3 again. The Shopping Cart microservice uses product information that it gets by querying the Product Catalog microservice. How long the product catalog information for any given product is likely to be correct is best decided by Product Catalog, which owns the data. Therefore, Product Catalog should add cache headers to its responses, and Shopping Cart should use them to decide how long it can cache a response. Figure 6.4 shows a sequence of requests to Product Catalog that Shopping Cart wants to make.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

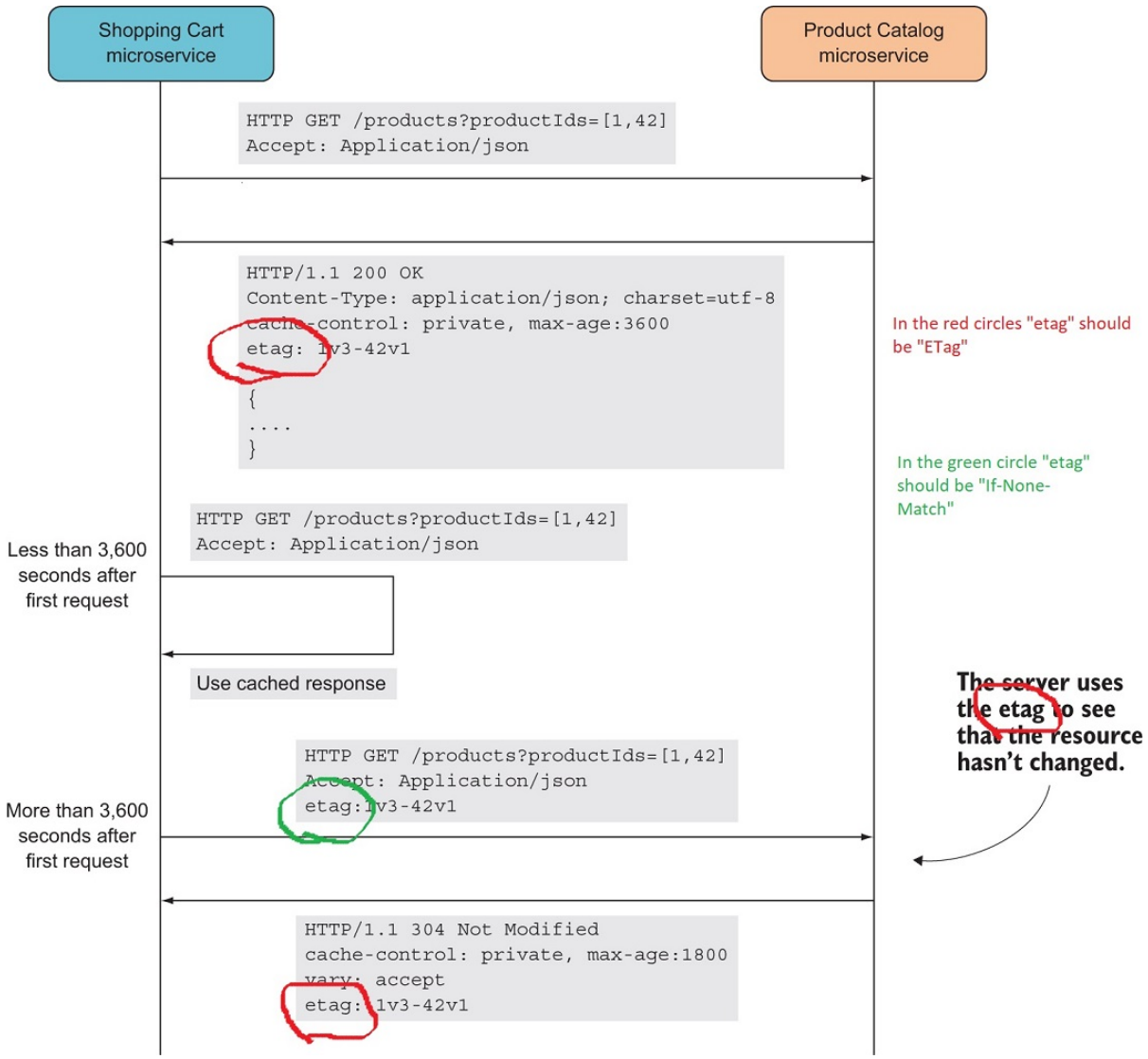

Figure 6.4 The Product Catalog microservice can allow its collaborators to cache responses by including cache headers in its HTTP responses. In this example, it sets max-age to indicate how long responses may be cached, and it also includes an etag built from the product IDs and versions.

In figure 6.4, the cache headers on the response to the first request tell the Shopping Cart microservice that it can cache the response for 3,600 seconds. The second time Shopping Cart wants to make the same request, the cached response is reused because fewer than 3,600 seconds have passed. The third time, the request to the Product Catalog microservice is made because more than 3,600 seconds have passed. That request includes the `ETag` from the first response in the `If-None-Match` header. Product Catalog uses the `ETag` to decide that the response would still be the same, so it sends back the shorter 304 Not Modified response instead of a full response. The 304 response includes a new set of cache headers that allows Shopping Cart to cache the already-cached response for an additional 1,800 seconds.

In the context of microservices running relatively close to each other - e.g. within the same data center - eliminating requests for information the caller already by using `cache-control` headers

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

is often sufficient to get the speed and robustness benefits we are after. The need to also use `ETags` to eliminate unnecessary response bodies arises in situations where the responses to queries are big and therefore require significant bandwidth or in when the distance between the microservices is larger - e.g. they are in different data centers, or in different zones in a cloud.

In sections 6.3.3 and 6.3.4, we'll discuss how to include `cache-control` headers in responses from action methods in our controllers. We'll also look at reading them on the client side from the response.

USING READ MODELS TO MIRROR DATA OWNED BY OTHER MICROSERVICES

It's normal for a microservice to query its own database for data it owns, but querying its database for data it doesn't own may not seem as natural. The natural way to get data owned by another microservice may seem to be to query that microservice. But it's often possible to replace a query to another microservice with a query to the microservice's own database by creating a *read model*: a data model that can be queried easily and efficiently. This is in contrast to the model used to store the data owned by the microservice, where the purpose is to store an authoritative copy of the data and to be able to easily update it when necessary.

Data is, of course, also written to read models—otherwise they'd be empty—but the data is written as a consequence of changes somewhere else. You trade some additional complexity at write time for less complexity at read time.

Read models are often based on events from other microservices. One microservice subscribes to events from another microservice and updates its own model of the event data as events arrive.

Read models can also be built from responses to queries to other microservices. In this case, the lifetime of the data in the read model is decided by the cache headers on those responses, just as in a straight cache of the responses. The difference between a straight cache and a read model is that to build a read model, the data in the responses is transformed and possibly enriched to make later reads easy and efficient. This means the shape of the data is determined by the scenarios in which it will be read instead of the scenario in which it was written.

Let's consider an example. The Shopping Cart microservice publishes events every time an item is added to or removed from a shopping cart. Figure 6.5 shows a Shopper Tracking microservice that subscribes to those events and updates a read model based on the events. Shopper Tracking allows business users to query how many times specific items are added to or removed from shopping carts.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

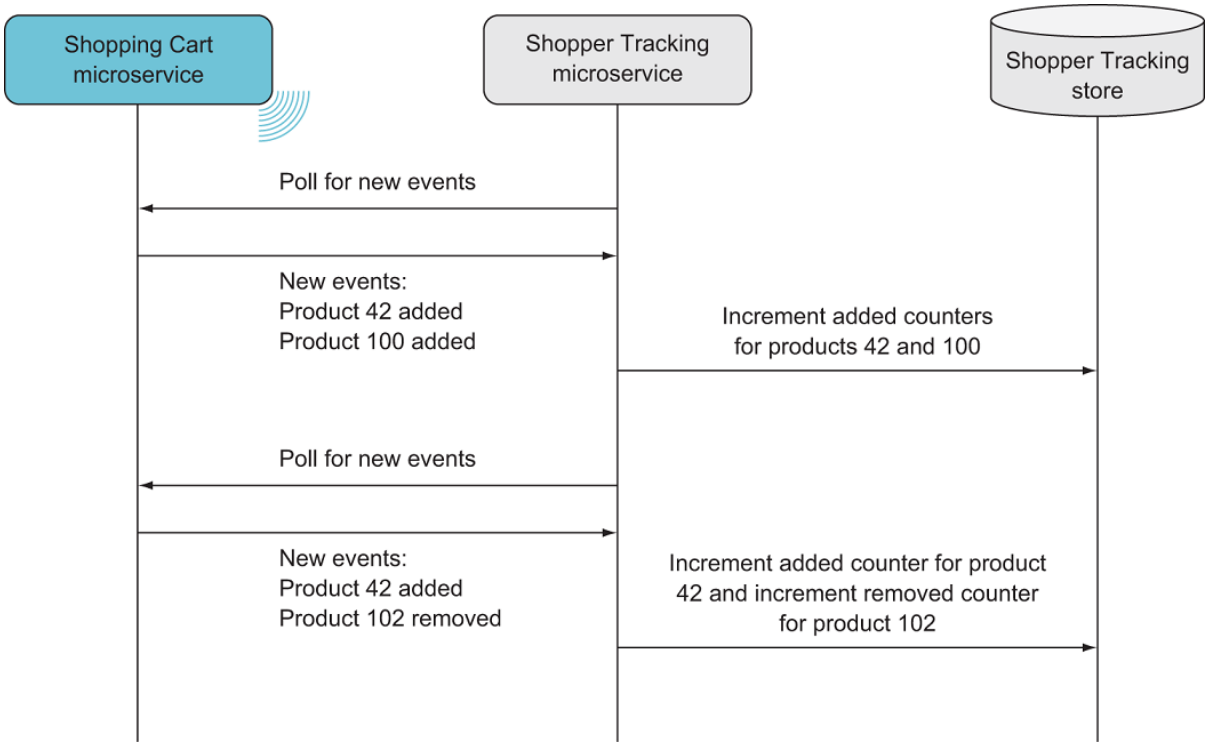

Figure 6.5 The Shopper Tracking microservice subscribes to events from the Shopping Cart microservice and keeps track of how many times products are added to or removed from shopping carts.

The events published from the Shopping Cart microservice aren't in themselves an efficient model to query when you want to find out how often a product has been added to or removed from shopping carts. But the events are a good source from which to build such a model. The Shopper Tracking microservice keeps two counters for every product: one for how many times the product has been added to a shopping cart, and one for how many times it's been removed. Every time an event is received from Shopping Cart, one of the counters is updated; and every time a query is made about a product, the two counters for that product are read.

6.2.3 Where does a microservice store its data?

A microservice can use one, two, or more databases. Some of the data stored by the microservice may fit well into one type of database, and other data may fit better into another type. Many viable database technologies are available, and I won't get into a comparison here. There are, however, some broad database categories that you can consider when you're making a choice, including relational databases, key/value stores, document databases, column stores, and graph databases.

If your production environment is in one the major public clouds there are some very compelling database services that you can take advantage of. These offerings differ in terms of database category - relation databases, document database and so on - but also in terms of operational characteristics. Some are "serverless" and require very little in terms of maintenance, others are more traditional. These should certainly also be taken into consideration.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

The choice of database technology (or technologies) for a microservice can be influenced by many factors, including these:

- What shape is your data? Does it fit well into a relational model, a document model, or a key/value store, or is it a graph?
- What are the write scenarios? How much data is written? Do the writes come in bursts, or are they evenly distributed over time?
- What are the read scenarios? How much data is read at a time? How much is read altogether? Do the reads come in bursts?
- How much data is written compared to how much is read?
- Which databases do the team already know how to develop against *and* run in production?

Asking yourself these questions—and finding the answers—will not only help you decide on a suitable database but will also likely deepen your understanding of the nonfunctional qualities expected from the microservice. You'll learn how reliable the microservice must be, how much load it must handle, what the load looks like, how much latency is acceptable, and so on.

Gaining that deeper understanding is valuable, but note that I'm not recommending that you undertake a major analysis of the pros and cons of different databases each time you spin up a new microservice. You should be able to get a new microservice going and deployed to production quickly. The goal isn't to find a database technology that's perfect for the job—you just want to find one that's suitable given your answers to the previous questions. You may be faced with a situation in which a document base seems like a good choice and in which you're confident that both Couchbase and MongoDB would be well suited. In that case, choose one of them. It's better to get the microservice to production with one of them quickly and at a later stage possibly replace the microservice with an implementation that uses the other, than it is to delay getting the first version of the microservice to production because you're analyzing Couchbase and MongoDB in detail.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **How many database technologies in the system?**

The decision about which database you should use in a microservice isn't solely a matter of what fits well in that microservice. You need to take the broader landscape into consideration. In a microservice system, you'll have many microservices and many data stores. It's worth considering how many different database technologies you want to have in the system. There's a trade-off between standardizing on a few database technologies and having a free-for-all.

On the side of standardizing are goals like these:

- Running the databases reliably in production and continuing to do so in the long run.
- Developers being able to get into and work effectively in the codebase of a microservice they haven't touched before.

Favoring a free-for-all are these types of goals:

- Being able to choose the optimal database technology for each microservice in terms of maintainability, performance, security, reliability, and so on.
- Keeping microservices replaceable. If new developers take over a microservice and don't agree with the choice of database, they should be able to replace the database or even the microservice as a whole.

How these goals are weighed against each other changes from organization to organization. It's important to be aware that there are trade-offs, but I will go as far as recommending some standardization within an organization.

6.3 Implementing data storage in a microservice

We've discussed where data should go in a microservice system, including which data a microservice should own and which data it should mirror. It's time to switch gears and look at the code required to store the data.

I'll focus on how a microservice can store the data it owns, including how to store the events it raises. I'll first show you how to do this using SQL Server and the lightweight Dapper data access library. Then I'll show you how to store events in a database specifically designed for storing events—the aptly named Event Store database.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **New technologies used in this chapter**

In this chapter, you'll begin using a couple of technologies that you haven't used yet in this book:

- *SQL Server*— Microsoft's SQL database. For purposes of this chapter many different databases would work fine. I am choosing SQL Server because I suspect it will be familiar to many readers.
- *Dapper* (https://github.com/StackExchange/dapper-dot-net)— A lightweight object-relational mapper (ORM). I'll introduce Dapper next.
- *Event Store* (https://geteventstore.com)— A database product specifically designed to store events. I'll introduce Event Store in a moment.

SIDEBAR **Dapper: a lightweight O/RM**

Dapper is a simple library for working with data in a SQL database from C#. It's part of a family of libraries sometimes referred to as *micro ORMs*, which also includes Simple.Data and Massive. These libraries focus on being simple to use and fast, and they embrace SQL.

Whereas a more traditional ORM writes all the SQL required to read data from and write it to the database, Dapper expects you to write your own SQL. I find this to be liberating when dealing with a database with a simple schema.

In the spirit of choosing lightweight technologies for microservices, I choose to use Dapper over a full-fledged ORM like Entity Framework or NHibernate. Often the database for a microservice is simple, and in such cases I find it easiest to add a thin layer—like Dapper—on top of it for a simpler solution overall. I could have chosen to use any of the other micro ORMs, but I like Dapper. In this chapter, you'll use Dapper to talk to SQL Server, but Dapper also works with other SQL databases like PostgreSQL, MySQL, or Azure SQL.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Event Store: a dedicated event database**

Event Store is an open source database server designed specifically for storing events. Event Store stores events as JSON documents, but it differs from a document database by assuming that the JSON documents are part of a stream of events. Although Event Store is a niche product because it's so narrowly focused on storing events, it's in widespread use and has proven itself in heavy-load production scenarios.

In addition to storing events, Event Store has facilities for reading and subscribing to events. For instance, Event Store exposes its own event feeds—as ATOM feeds—that clients can subscribe to. If you don't mind depending on Event Store, using its ATOM event feed to expose events to other microservices can be a viable alternative to the way you'll implement event feeds in this book.

Event Store works by exposing an HTTP API for storing, reading, and subscribing to events. There are a number of Event Store client libraries in various languages—including C#, F#, Java, Scala, Erlang, Haskell, and JavaScript—that make it easier to work with the database.

6.3.1 Preparing a development setup

Before we dive into implementation we will need run a SQL Server on localhost which we will do in a Docker container.

First pull down the latest SQL Server docker image to your machine

```
docker pull mcr.microsoft.com/mssql/server
```

and then run it:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=yourStrong(!)Password' -p 1433:1433 -d mcr.microsoft.com/m
```

lastly confirm that SQL is indeed running by listing locally running container and checking that SQL Server is in the list:

```
docker ps

CONTAINER ID        IMAGE                             COMMAND                  CREATED             STATUS
3388b710892f        mcr.microsoft.com/mssql/server    "/opt/mssql/bin/perm…"   52 minutes ago      Up 52
```

With this in place we are ready to start implementing data storage in our microservices.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

6.3.2 Storing data owned by a microservice

Once you've decided which data a microservice owns, storing that data is relatively straightforward. The details of how it's done depend on your choice of database. The only difference specific to microservices is that the data store is solely owned and accessed by the microservice itself.

As an example, let's go back to the Shopping Cart microservice.

It owns the users' shopping carts and therefore stores them. You'll store the shopping carts in SQL Server using Dapper.

You implemented most of the Shopping Cart microservice in chapter 2. Here, you'll fill in the data store bits.

> **NOTE** We are using SQL Server as an example but everything we do with SQL Server could just as well be done with any other relation database, be it PostgreSQL, MySQL or a cloud hosted relational database like Azure SQL or AWS Aurora.

If you're familiar with storing data in SQL Server, the implementation should be no surprise, and that's the point. Storing the data owned by a microservice doesn't need to involve anything fancy. These are the steps for storing the shopping cart:

1. Create a database.
2. Use Dapper to implement the code to read, write, and update shopping carts.

First, you'll create a simple database for storing shopping carts. It will have two tables, as shown in figure 6.6.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

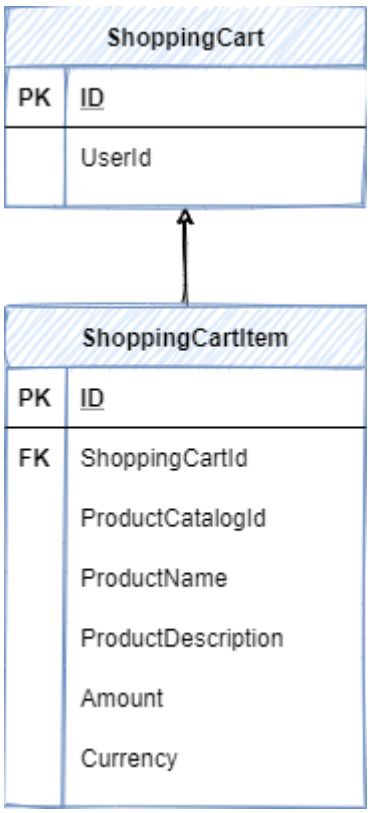

Figure 6.6 ShoppingCart has only two tables: one has a row for each shopping cart, and the other has a row per item in a shopping cart.

The ShoppingCart database and the two tables - ShoppingCart and ShoppingCartItem - can be created with this SQL scripts which you can execute with various tools including SQL Management Studio and Visual Studio Code:

```
CREATE DATABASE ShoppingCart
GO

USE [ShoppingCart]
GO

CREATE TABLE [dbo].[ShoppingCart](
  [ID] int IDENTITY(1,1) PRIMARY KEY,
  [UserId] [bigint] NOT NULL,
  CONSTRAINT ShoppingCartUnique UNIQUE([ID], [UserID])
)
GO

CREATE INDEX ShoppingCart_UserId
ON [dbo].[ShoppingCart] (UserId)
GO

CREATE TABLE [dbo].[ShoppingCartItem](
  [ID] int IDENTITY(1,1) PRIMARY KEY,
    [ShoppingCartId] [int] NOT NULL,
    [ProductCatalogId] [bigint] NOT NULL,
    [ProductName] [nvarchar](100) NOT NULL,
    [ProductDescription] [nvarchar](500) NULL,
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

```
    [Amount] [int] NOT NULL,
    [Currency] [nvarchar](5) NOT NULL
)

GO

ALTER TABLE [dbo].[ShoppingCartItem]  WITH CHECK ADD CONSTRAINT [FK_ShoppingCart] FOREIGN KEY([ShoppingC
REFERENCES [dbo].[ShoppingCart] ([Id])
GO

ALTER TABLE [dbo].[ShoppingCartItem] CHECK CONSTRAINT [FK_ShoppingCart]
GO

CREATE INDEX ShoppingCartItem_ShoppingCartId
ON [dbo].[ShoppingCartItem] (ShoppingCartId)
GO
```

With the database in place, you can implement the code in the Shopping Cart microservice that reads, writes, and updates the database. You'll install the Dapper NuGet package into the microservice. Remember that you do this by adding Dapper to the ShoppingCart project with the `dotnet add package dapper` command from a command line in the same folder as the `ShoppingCart.csproj` file. After adding the Dapper package the item group with package references in the `ShoppingCart.csproj` file should look like this:

```
<ItemGroup>
  <PackageReference Include="Dapper" Version="2.0.35" />     ❶
  <PackageReference Include="Microsoft.Extensions.Http.Polly" rsion="3.1.0" />
  <PackageReference Include="Polly" Version="7.2.0" />
  <PackageReference Include="Scrutor" Version="3.1.0" />
</ItemGroup>
```

❶ Adds the Dapper library

In chapter 2, the Shopping Cart microservice was expecting an implementation of an `IShoppingCart` interface. You'll change that interface slightly to allow the implementation of it to make asynchronous calls to the database. This is the modified interface:

```
public interface IShoppingCartStore
{
  Task<ShoppingCart> Get(int userId);
  Task Save(ShoppingCart shoppingCart);
}
```

Now it's time to look at the implementation of the `IShoppingCartStore` interface. First, let's consider the code for reading a shopping cart from the database.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.1 Reading shopping carts with Dapper

```
namespace ShoppingCart.ShoppingCart
{
  using System.Data;
  using System.Data.SqlClient;
  using System.Linq;
  using System.Threading.Tasks;
  using Dapper;

  public interface IShoppingCartStore
  {
    Task<ShoppingCart>Get(int userId);
    Task Save(ShoppingCart shoppingCart);
  }

  public class ShoppingCartStore : IShoppingCartStore
  {
    private string connectionString =
      @"Data Source=localhost;Initial Catalog=ShoppingCart;                    ❶
User Id=SA; Password=yourStrong(!)Password";

    private const string readItemsSql =
      @"
select ShoppingCart.ID, ProductCatalogId,                                    ❷
ProductName, ProductDescription, Currency, Amount
from ShoppingCart, ShoppingCartItem
where ShoppingCartItem.ShoppingCartId = ShoppingCart.ID
and ShoppingCart.UserId=@UserId";

    public async Task<ShoppingCart> Get(int userId)
    {
      await using var conn = new SqlConnection(this.connectionString);       ❸
      var items = (await
          conn.QueryAsync(                                                   ❹
            readItemsSql,
            new {UserId = userId}))
        .ToList();
      return new ShoppingCart(
        items.FirstOrDefault()?.ID,
        userId,
        items.Select(x =>                                                    ❺
          new ShoppingCartItem(
            (int) x.ProductCatalogId,
            x.ProductName,
            x.ProductDescription,
            new Money(x.Currency, x.Amount))));
    }
  }
}
```

❶ Connection string to the ShoppingCart database in the MS SQL Docker container

❷ Dapper expects and allows you to write your own SQL

❸ Opens a connection to the ShoppingCart database

❹ Uses a Dapper (extension method, Dapperextension method to execute a SQL query

❺ The result set from the SQL query to ShoppingCartItem

Dapper is a simple tool that provides some convenient extension methods on `IDbConnection` to

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

make working with SQL in C# easier. It also provides some basic mapping capabilities. For instance when the rows returned by a SQL query have column names equal to the property names in a class Dapper can map automatically to instances of the class.

Dapper doesn't try to hide the fact that you're working with SQL, so you see SQL strings in the code. This may feel like a throwback to the earliest days of .NET. I find that as long as I'm working with a simple database schema—as I usually am in micro-services—the SQL strings in C# code aren't a problem.

Writing a shopping cart to the database is also done through Dapper. The implementation is the following method in `ShoppingCartStore`.

Listing 6.2 Writing shopping carts with Dapper

```
    private const string insertShoppingCartSql =
@"insert into ShoppingCart (UserId) OUTPUT inserted.ID VALUES (@UserId)";

    private const string deleteAllForShoppingCartSql =
@"delete item from ShoppingCartItem item
inner join ShoppingCart cart on item.ShoppingCartId = cart.ID
and cart.UserId=@UserId";

    private const string addAllForShoppingCartSql =
@"insert into ShoppingCartItem
(ShoppingCartId, ProductCatalogId, ProductName,
ProductDescription, Amount, Currency)
values
(@ShoppingCartId, @ProductCatalogId, @ProductName,
@ProductDescription, @Amount, @Currency)";

    public async Task Save(ShoppingCart shoppingCart)
    {
      await using var conn = new SqlConnection(this.connectionString);
      await conn.OpenAsync();
      await using (var tx = conn.BeginTransaction())
      {
        var shoppingCartId =
          shoppingCart.Id ??
            await conn.QuerySingleAsync<int>(                          ❶
              insertShoppingCartSql,
              new {shoppingCart.UserId}, tx);

        await conn.ExecuteAsync(                                       ❷
          deleteAllForShoppingCartSql,
          new {UserId = shoppingCart.UserId},
          tx);
        await conn.ExecuteAsync(                                       ❸
          addAllForShoppingCartSql,
          shoppingCart.Items.Select(x =>
            new
            {
              shoppingCartId,
              x.ProductCatalogId,
              Productdescription = x.Description,
              x.ProductName,
              x.Price.Amount,
              x.Price.Currency
            }),
          tx);
      }
    }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

1. Create a row in the ShoppingCart table if the shopping cart does not already have an id
2. Deletes all preexisting shopping cart items
3. Adds the current shopping cart items

That concludes the the code that stores shopping cart information in the Shopping Cart microservice. It's similar to storing data in a more traditional setting—like a monolith or traditional SOA service—except that the narrow scope of a micro-service means the model is often so simple that little-to-no mapping between C# code and a database schema is needed.

6.3.3 Storing events raised by a microservice

This section looks at storing the events raised by a microservice. During command processing, a microservice can decide to raise events. Figure 6.7 shows the standard set of components in a microservice: the domain model raises the events. It typically does so when there's a change or a set of changes to the state of the data for which the microservice is responsible.

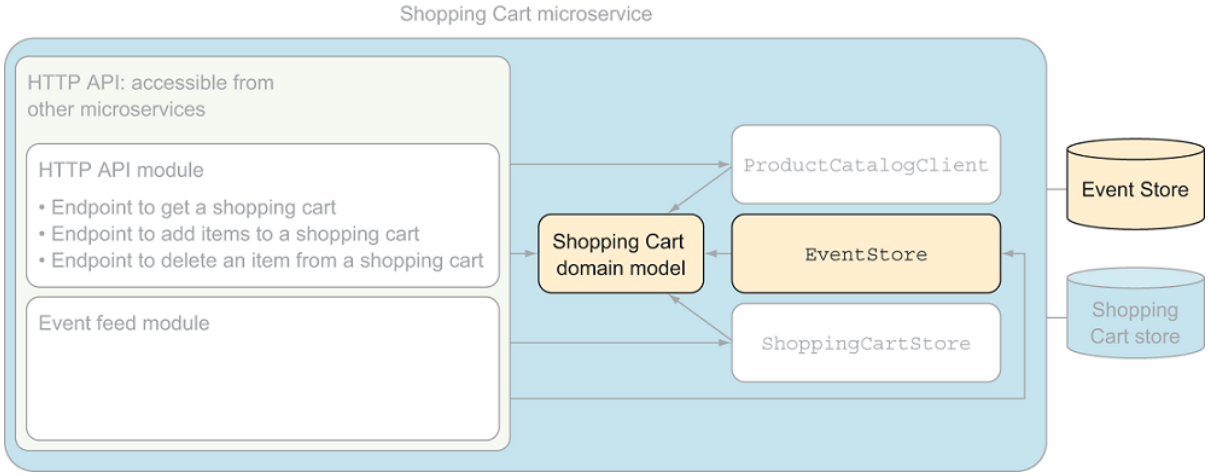

Figure 6.7 The components in the Shopping Cart microservice involved in raising and saving events are the Shopping Cart domain model, the EventStore component, and the Event Store database.

The events should reflect a change to the state of the data owned by the microservice. The events should also make sense in terms of the capability implemented by the microservice. For example, in a Shopping Cart microservice, when a user has added an item to their shopping cart, the event raised is `ItemAddedToShoppingCart`, not `RowAddedToShoppingCartTable`. The difference is that the first signifies an event of significance to the system—a user did something that's interesting in terms of the business—whereas the latter would report on a technical detail—a piece of software did something because a programmer decided to implement it that way. The events should be of significance at the level of abstraction of the capability

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

implemented by the microservice, and they will often cover several updates to the underlying database. The events should correspond to business-level transactions, not to database transactions.

Whenever the domain logic in a microservice raises an event, it's stored to the event store in the microservice. In figure 6.8, this is done through the `EventStore` component, which is responsible for talking to the database where the events are stored.

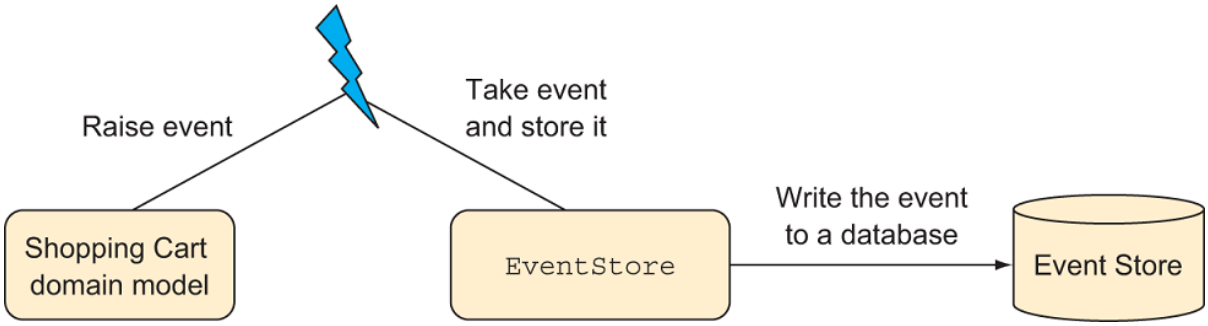

Figure 6.8 When the domain model raises an event, the EventStore component code must write it to the Event Store database.

The following two sections show two implementations of an `EventStore` component. The first stores the events by hand to a table in a SQL database, and the second uses the open source Event Store database.

STORING EVENTS BY HAND

Here, you'll build an implementation of the `EventStore` component in the Shopping Cart microservice that stores events to a table in SQL Server. The `EventStore` component is responsible for both writing events to and reading them from that database.

The following steps are involved in implementing the `EventStore` component:

1. Add an `EventStore` table to the `ShoppingCart` database. This table will contain a row for every event raised by the domain model.
2. Use Dapper to implement the writing part of the `EventStore` component.
3. Use Dapper to implement the reading part of the `EventStore` component.

Before we dive into implementing the `EventStore` component, here's a reminder of what the `Event` type in Shopping Cart looks like:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.3 The event type in the ShoppingCart microservice

```
public struct Event
{
  public long SequenceNumber { get; }
  public DateTimeOffset OccurredAt { get; }
  public string Name { get; }
  public object Content { get; }

  public Event(
    long sequenceNumber,
    DateTimeOffset occurredAt,
    string name,
    object content)
  {
    this.SequenceNumber = sequenceNumber;
    this.OccurredAt = occurredAt;
    this.Name = name;
    this.Content = content;
  }
}
```

It's events of this type that you'll store in the event store database. The first step is to go into the `ShoppingCart` database and add a table like the one shown in figure 6.9. The database script in Chapter05\ShoppingCart\src\database-scripts\create-shopping-cart-db.sql in the code download creates this table, along with the other two tables in the `ShoppingCart` database.

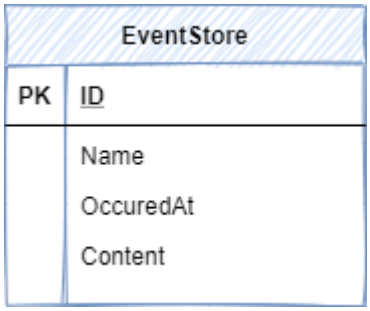

Figure 6.9 The EventStore table has four columns for these categories: event ID, event name, the time the event occurred, and the contents of the event.

Next, add a file named EventStore.cs to Shopping Cart, and add to it the following code for writing events.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.4 Raising an event, which amounts to storing it

```
namespace ShoppingCart.EventFeed
{
  using System;
  using System.Collections.Generic;
  using System.Data.SqlClient;
  using System.Linq;
  using System.Threading.Tasks;
  using Dapper;
  using Newtonsoft.Json;

  public interface IEventStore
  {
    Task<IEnumerable<Event>> GetEvents(long firstEventSequenceNumber,
      long lastEventSequenceNumber);
    Task Raise(string eventName, object content);
  }

  public class EventStore : IEventStore
  {
    private string connectionString =
      @"Data Source=localhost;Initial Catalog=ShoppingCart;
User Id=SA; Password=yourStrong(!)Password";

    private const string writeEventSql =
@"insert into EventStore(Name, OccurredAt, Content)
values (@Name, @OccurredAt, @Content)";

    public async Task Raise(string eventName, object content)
    {
      var jsonContent = JsonConvert.SerializeObject(content);
      await using var conn = new SqlConnection(this.connectionString);
      await conn.ExecuteAsync(                                          ❶
        writeEventSql,
        new
        {
          Name = eventName,
          OccurredAt = DateTimeOffset.Now,
          Content = jsonContent
        });
    }
  }
```

❶ Uses Dapper to execute a simple SQL insert statement

This code doesn't compile yet, because the `IEventStore` interface has another method: one for reading events. That side is implemented as shown next.

NOTE Storing events essentially amounts to storing a JSON serialization of the content of the event in a row in the `EventStore` table along with the ID of the event, the name of the event, and the time at which the event was raised. The concept of storing events and publishing them through an event feed may be new, but the implementation is pretty simple.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.5 `EventStore` method for reading events

```
private const string readEventsSql =
  @"select * from EventStore where ID >= @Start and ID <= @End";

public async Task<IEnumerable<Event>> GetEvents(
  long firstEventSequenceNumber,
  long lastEventSequenceNumber)
{
  await using var conn = new SqlConnection(this.connectionString);
  return await conn.QueryAsync<Event>(                                    ❶
      readEventsSql,
      new
      {
        Start = firstEventSequenceNumber,                                 ❷
        End = lastEventSequenceNumber
      });
}
```

❶ Maps EventStore table rows to Event objects

❷ Reads EventStore table rows between start and end

That's all you need to implement a basic event store. The Shopping Cart microservice can now raise events in the domain model and rely on the `EventStore` component to write them to the `EventStore` table in the `ShoppingCart` database. Furthermore, Shopping Cart has an event feed that you implemented back in chapter 2, which now uses the `EventStore` component to read events from the database. It will send them to event subscribers when they poll the feed.

This event store implementation is very basic and is not ready for all production use cases. For instance, it may run into lock-contention problems as when the microservice starts raising events from several concurrent threads especial under high load. This example does, however, show what it means to store events. .

NOTE There are several high quality open source projects that implement an event store on top of a relational database including SqlStreamStore and Marten.

STORING EVENTS USING THE EVENT STORE DATABASE SYSTEM

You'll now implement another version of the `EventStore` component in the Shopping Cart microservice, this time using the open source Event Store database. The advantage of using Event Store over storing events in SQL Server is that its API is geared specifically toward storing events, reading events, and subscribing to new events. Event Store is an open source, mature, well-tested event store implementation that can scale and run stably under load. Furthermore, it comes with some nice added features out the box, such as a web interface for inspecting events and Atom event feeds. SQL Server is, of course, also mature, well-tested, scalable, and stable, but it isn't specifically geared toward storing events.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

You'll implement this version with the following steps. When you're finished, you'll have a fully working implementation of the `EventStore` component in the Shopping Cart microservice based on the Event Store database:

1. Run Event Store database in a Docker container.
2. Write events to Event Store via the `EventStore` component.
3. Read events from Event Store via the `EventStore` component.

> **NOTE** You can learn much more about Event Store than we will cover in this chapter at https://eventstore.com/

You can pull the Event Store Docker image down with this command:

```
docker pull eventstore/eventstore
```

Once pulled down you can run the Event Store container like this:

```
docker run --name eventstore-node -it -p 2113:2113 -p 1113:1113 --rm eventstore/eventstore
```

You can check whether the Event Store database is running by going to http://127.0.0.1:2113/. You should see a login prompt that lets you log in with the user name `admin` and the password `changeit`.

In order to use the Event Store database from the Shopping Cart microservice code, you first need to add the `EventStore.Client` NuGet package to the project. With that installed, you can implement the `EventStore` component against the Event Store database. The following listing shows the code for writing events.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.6 Storing events to the Event Store database

```
namespace ShoppingCart.EventFeed
{
  using System;
  using System.Collections.Generic;
  using System.Linq;
  using System.Text;
  using System.Threading.Tasks;
  using EventStore.ClientAPI;
  using Newtonsoft.Json;

  public class EsEventStore : IEventStore
  {
    private const string ConnectionString =
      "tcp://admin:changeit@localhost:1113";

    public async Task Raise(string eventName, object content)
    {
      using var connection =
        EventStoreConnection.Create(new Uri(ConnectionString));   ❶
      await connection.ConnectAsync();                            ❷
      await connection.AppendToStreamAsync(                       ❸
        "ShoppingCart",
        ExpectedVersion.Any,
        new EventData(                                            ❹
          Guid.NewGuid(),
          "ShoppingCartEvent",
          isJson: true,
          data: Encoding.UTF8.GetBytes(
            JsonConvert.SerializeObject(content)),
          metadata: Encoding.UTF8.GetBytes(
            JsonConvert.SerializeObject(new EventMetadata
          {
            OccuredAt = DateTimeOffset.UtcNow,                    ❺
            EventName = eventName
          })));
    }

    public class EventMetadata
    {
      public DateTimeOffset OccuredAt { get; set; }
      public string EventName { get; set; }
    }
  }
}
```

❶ Creates a connection to EventStore

❷ Opens the connection to EventStore

❸ Writes the event to EventStore

❹ EventData is EventStore's representation of an event.

❺ Maps OccurredAt and EventName to metadata to be stored along with the event

This code maps the Shopping Cart microservice's own `Event` type to the Event Store database's `EventData` type, and then stores that to the Event Store database. The implementation for reading events back from the Event Store database is shown next.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.7 Reading events from the Event Store database

```
public async Task<IEnumerable<Event>>
 GetEvents(long firstEventSequenceNumber, long lastEventSequenceNumber)
{
  using var connection =
    EventStoreConnection.Create(new Uri(ConnectionString));
  await connection.ConnectAsync();
  var result = await connection.ReadStreamEventsForwardAsync(                ❶
    "ShoppingCart",
    start: firstEventSequenceNumber,
    count: (int) (lastEventSequenceNumber - firstEventSequenceNumber),
    resolveLinkTos: false);
  return result.Events                                                       ❷
    .Select(e =>
      new
      {
        Content = Encoding.UTF8.GetString(e.Event.Data),                     ❸
        Metadata = JsonConvert.DeserializeObject<EventMetadata>(             ❹
          Encoding.UTF8.GetString(e.Event.Metadata))
      })
    .Select((e, i) =>                                                        ❺
      new Event(
        i + firstEventSequenceNumber,
        e.Metadata.OccuredAt,
        e.Metadata.EventName,
        e.Content));
}
```

❶ Reads events from the Event Store

❷ Accesses the events on the result from the Event Store

❸ Gets the content part of each event

❹ Gets the metadata part of each event

❺ Maps to events from Event Store Event objects

This code reads the events from Event Store, deserializes the content and metadata parts of the events, and then maps them back to the Shopping Cart Microservice's own `Event` type.

This completes the implementation of the `EventStore` component based on the Event Store database. Let's now look at using caching.

6.3.4 Setting cache headers in HTTP responses

Let's consider the microservices in figure 6.2 again. The Shopping Cart microservice uses product information that it gets by querying the Product Catalog microservice; you implemented the Shopping Cart microservice part of that collaboration in chapter 2. Here, you'll first set cache headers in the code implementing the endpoint in Product Catalog. Then, you'll rewrite the code in Shopping Cart that calls Product Catalog to read and use the cache header.

Assume that the /products endpoint in the Product Catalog microservice is implemented in an MVC Controller called `ProductsController`. You'll take a comma-separated list of product

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

IDs as a query parameter. The endpoint returns the product information for each of the products identified by that list of product IDs. The implementation is similar to the MVC controllers you've already seen in this book. The new part is that you'll use the `ResponseCache` attribute to add a `cache-control` header to the response that allows clients to cache the response for 24 hours.

Listing 6.8 Adding cache headers to the product list

```
namespace ProductCatalog
{
  using System.Collections.Generic;
  using System.Linq;
  using Microsoft.AspNetCore.Mvc;

  [Route("/products")]
  public class ProductCatalogController : Controller
  {
    private readonly IProductStore productStore;

    public ProductCatalogController(IProductStore productStore) =>
      this.productStore = productStore;

    [HttpGet("")]
    [ResponseCache(Duration = 86400)]                                    (1)
    public IEnumerable<ProductCatalogProduct> Get()
    {
      var productIds = ParseProductIdsFromQueryString(
        this.Request.Query["productIds"]);
      var products = this.productStore.GetProductsByIds(productIds);
      return products;
    }

    private static IEnumerable<int>
      ParseProductIdsFromQueryString(string productIdsString) => ...
  }
```

(1) Adds a cache-control header, with max-age in seconds

This implementation adds a cache-control header to the response that looks like this:

```
cache-control: public,max-age:86400
```

The header tells callers that the response may be cached for as long as indicated by `max`-value, which is given in seconds. In this case, callers may cache the response for 86,400 seconds (24 hours).

6.3.5 Reading and using cache headers

In chapter 2, you saw code make calls to the /products endpoint from the Shopping Cart microservice. That code is as follows; it's part of the `ProductCatalogClient` class.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.9 Calling the Product Catalog microservice

```
private async Task<HttpResponseMessage>
  RequestProductFromProductCatalog(int[] productCatalogIds)
{
  var productsResource =
    string.Format(getProductPathTemplate,
      string.Join(",", productCatalogIds));
  return await
    this.client.GetAsync(productsResource);
}
```

With this code, an HTTP request is made every time Shopping Cart needs product information, regardless of any cache headers. This is inefficient in cases where Shopping Cart needs information about the same products several times within 24 hours, because that's the `max-age` value set in the responses from the Product Catalog microservice. Such cases will occur every time a user adds to their shopping cart an item that another user has added to their shopping cart within the preceding 24 hours. That's likely to happen often.

Let's extend the code making the call to the /products endpoint in Product Catalog to take cache headers into account. Add a dependency to `ProductCatalogClient` on a cache that implements an `ICache` interface:

```
private readonly HttpClient client;
private readonly ICache cache;
private static string productCatalogueBaseUrl =  @"https://git.io/JeHiE";
private static string getProductPathTemplate = "?productIds=[{0}]";

public ProductCatalogClient(HttpClient client, ICache cache)
{
  client.BaseAddress = new Uri(productCatalogueBaseUrl);
  client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/json"));
  this.client = client;
  this.cache = cache;
}
```

As you'll recall, ASP.NET Core handles dependency injection for you, so as long as there's an implementation of the `ICache` interface in registered in the IServiceCollection, ASP.NET Core will inject it. Here, I'll only show the interface, but in the code download, you can find a simple static cache implementing the interface. The interface is straightforward and has two methods:

```
public interface ICache
{
  void Add(string key, object value, TimeSpan ttl);          (1)
  object Get(string key);
}
```

(1) "ttl" means time to live.

You'll use the `cache` variable on `ProductCatalogClient` to check whether there's a valid object in the cache before making an HTTP request.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 6.10 Making requests when there's no valid response in the cache

```
private async Task<HttpResponseMessage>
  RequestProductFromProductCatalog(int[] productCatalogIds)
{
  var productsResource = string.Format(
    getProductPathTemplate,
    string.Join(",", productCatalogIds));
  var response =
    this.cache.Get(productsResource) as HttpResponseMessage;       ❶
  if (response is null)                                            ❷
  {
    response = await this.client.GetAsync(productsResource);
    AddToCache(productsResource, response);
  }
  return response;
}

private void AddToCache(string resource, HttpResponseMessage response)
{
  var cacheHeader = response
    .Headers
    .FirstOrDefault(h => h.Key == "cache-control");                ❸
  if (!string.IsNullOrEmpty(cacheHeader.Key)
      && CacheControlHeaderValue.TryParse(                         ❹
        cacheHeader.Value.ToString(), out var cacheControl)
      && cacheControl.MaxAge.HasValue)
    this.cache.Add(resource, response, cacheControl.MaxAge.Value);  ❺
}
```

❶ Tries to retrieve a valid response from the cache

❷ Only makes the HTTP request if there's no response in the cache

❸ Reads the cache-control header from the response

❹ Parses the cache-control value and extracts max-age from it

❺ Adds the response to the cache if it has a max-age value

With this code in place in the Shopping Cart microservice, the responses from the Product Catalog microservice will be used for as long as the `max-age` value in the `cache-control` header allows (24 hours, in this example).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

6.4 Summary

- A microservice stores and owns all the data that belongs to the capability the microservice implements.
- A microservice is the authoritative source for the data it owns.
- A microservice stores its data in its own dedicated database.
- A microservice will often also cache data owned by other microservices for several reasons:
  - To reduce coupling to other microservices. This makes the overall system more stable.
  - To speed up processing by avoiding making remote calls.
  - To build up its own custom representations—known as read models—of data owned by another microservice to make its code simpler.
  - To build read models based on events from other microservices to avoid querying the other microservices, thus using an event-based collaboration style instead of a query-based one. Remember from chapter 5 that event-based collaboration is preferable because of the reduced coupling.
- Which database or databases a microservice uses is a design decision particular to that microservice. Different microservices can use different databases.
- Storing the data owned by a microservice is similar to storing data in other kinds of systems.
- You can use Dapper to read data from and write data to a SQL database.
- Storing events is essentially a matter of storing a serialized event to a database.
- A simple version of an event store involves storing events to a table in a SQL database.
- You can also implement an event store by storing events to the open source Event Store database, which is specifically designed to store events.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

7 Designing for robustness

In this chapter:

- Communicating robustly between microservices
- Letting the calling side take responsibility for robustness in the face of
- failure Rolling back versus rolling forward
- Implementing robust communication

This chapter introduces strategies for making a system of microservices robust in the face of failures. In general, whenever one microservice communicates with another microservice, the communication may fail. In this chapter, you'll learn about and implement some patterns for dealing with such failures. The strategies are fairly simple, yet they'll make the overall system much more robust.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Failures and errors**

I'll distinguish between the terms *failure* and *error*. A *failure* happens when something goes wrong in the system and the issue is caused by something outside the system. Some typical sources of failures are as follows:

- Lost network packets cause communication to fail.
- Lost connections cause communication to fail.
- Hardware failures cause microservices to fail.

An *error* happens when the system can't serve its users properly. Some typical examples of errors are these:

- A user sees an error page.
- The system hangs and never responds to a user action.
- The system gives back the wrong response to a user action.

Errors can stem from failures. On the other hand, failures only become errors if the software can't cope properly with failures. It follows that a perfect system would see failures, but no errors. Unfortunately, our systems aren't perfect, and we may as well accept that errors will occur, but learn to cope with them quickly.

7.1 Expect failures

When working with any nontrivial software system, you must expect failures to occur. Hardware can fail. Software may fail due to, for instance, unforeseen usage or corrupt data. A distinguishing factor of a microservice system is that there's a lot of communication between microservices. Figure 7.10 repeats the diagram from chapter 1 that shows the communication resulting from a user adding an item to a shopping cart. You see that just one user action results in quite a bit of communication—and a real system will likely have many concurrent users all performing many actions, and thus lots of communication going on. You must expect communication to fail from time to time. Communication between two microservices may not fail often, but looking at a microservice system as a whole, communication failures are likely to occur often due to the amount of communication.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

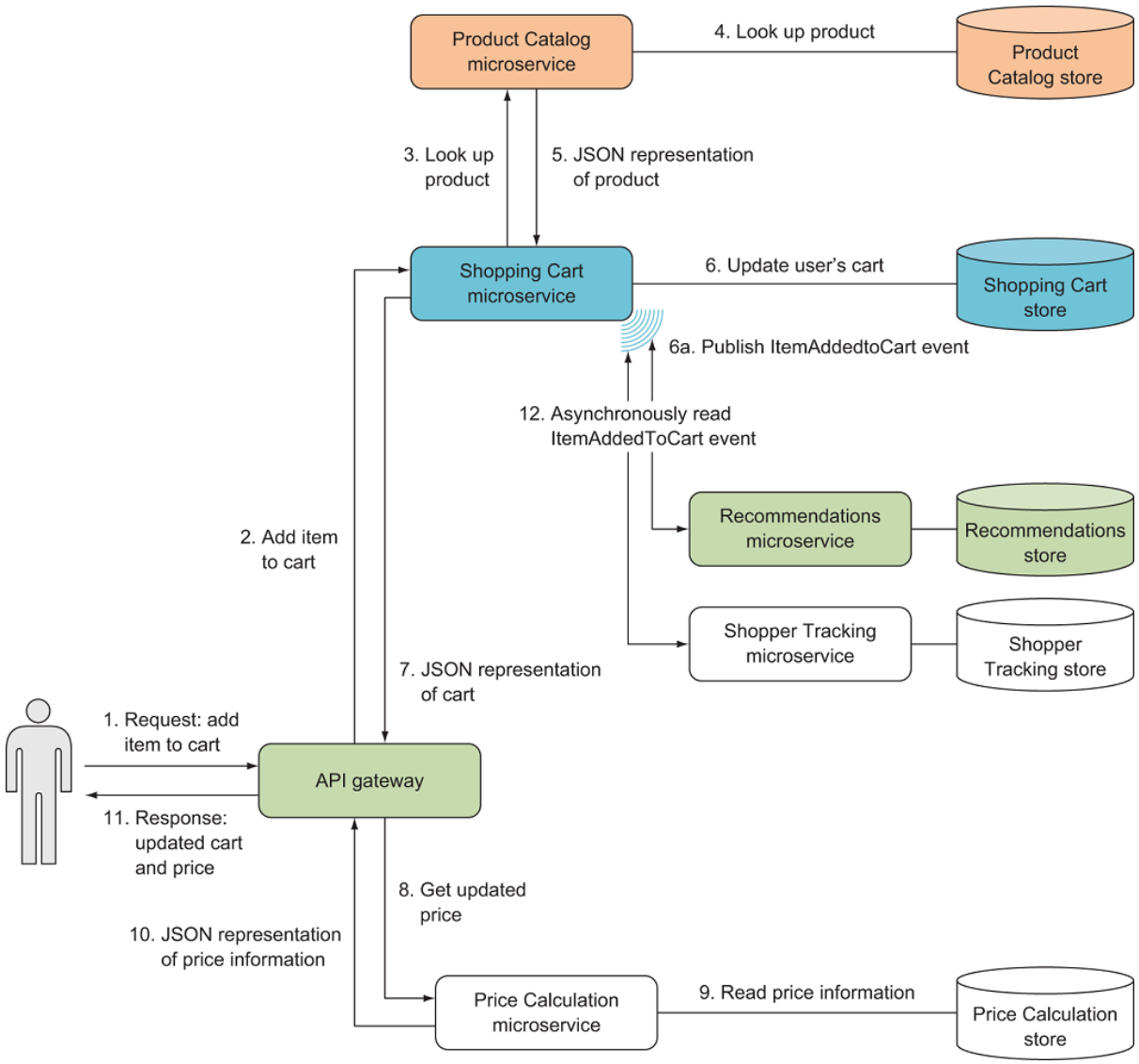

Figure 7.1 In a system of microservices, there will be many communication paths.

Because you have to expect that some of the communication in your microservice system will fail, you should design your microservices to be able to cope with those failures. As discussed in chapter 5, you can divide the collaborations between microservices into three categories: query-, command-, and event-based collaborations. When a communication fails, the impact depends on the type of collaboration and the way the microservices cope with it:

- *Query-based collaboration*—When a query fails, the caller doesn't get the information it needs. If the caller copes well with that, the system keeps working, but with degraded functionality. If the caller doesn't cope well, the result could be an error.
- *Command-based collaboration*—When sending a command fails, the sender can't know whether the receiver got the command. Again, depending on how the sender copes, this could result in an error, or it could result in degraded functionality.
- *Event-based collaboration*—When a subscriber polls an event feed, but the call fails, the impact is limited. The subscriber will poll the event feed again later and, assuming the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

event feed is up again, receive the events at that time. In other words, the subscriber will still get all events, but some of them will be delayed. This shouldn't be a problem for an event-based collaboration, because it's asynchronous anyway.

The following subsection discusses some important ways to prepare for handling failure well.

7.1.1 Keeping good logs

Once you accept that failures are bound to happen and that some of them may result not just in a degraded end user experience but also in errors, you must make sure you're able to understand what went wrong when an error occurs. That means you need good logs that allow you to trace what happened in the system and led to an error situation. "What happened" will often span several microservices, which is why you should introduce a central Log microservice, as shown in figure7.11; all the other microservices send log messages to it, and you can inspect and search the logs when you need to.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

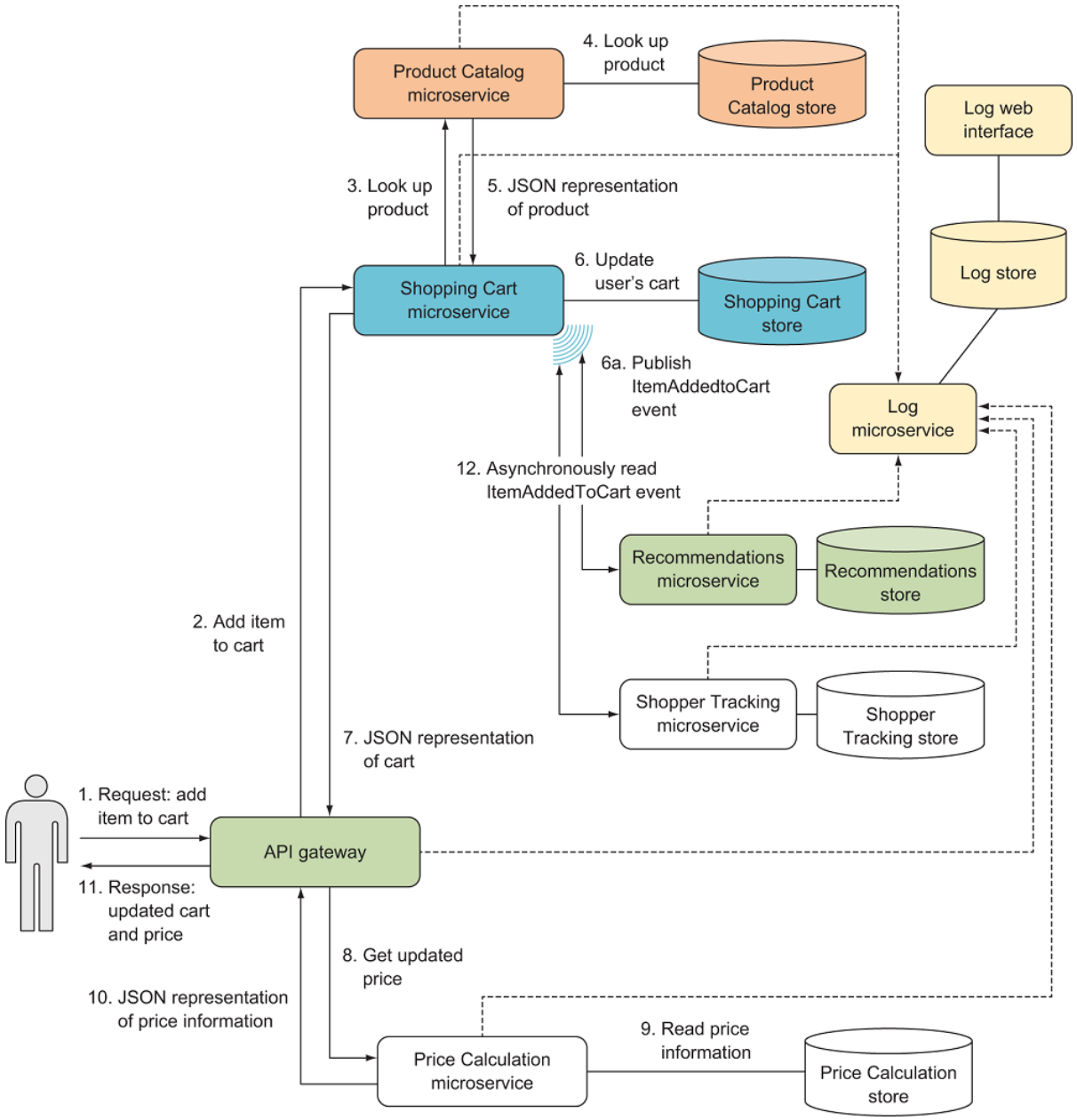

Figure 7.2 A central Log microservice receives log messages from all other microservices and stores them in a database or a search engine. The log data is accessible through a web interface. The dotted arrows shows microservices sending log messages to the central Log microservice.

The Log microservice is a central component that all other microservices use. You need to make certain that a failure in Log doesn't bring down the whole system by causing all other microservice to fail if they can't log messages. Therefore, sending log messages to Log must be decoupled so the microservice sending the message shouldn't wait for a response. That decoupling can be achieved in several different ways depending on the choices you make for your productions environment. When using containers the standard way to decouple from the Log microservices is for the other microservices to simply log to standard out and let a *log shipping* tool collect those logs and send them to the Log microservice.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Using an off-the-shelf solution for the Log microservice**

A central Log microservice doesn't implement a business capability of a particular system. It's an implementation of generic technical capability. In other words, the requirements for a Log microservice in system A aren't that different from the requirements for a Log microservice in system B. Therefore, I recommend using an off-the-shelf solution to implement your Log microservice. Products in this category include ELK, Datadog, Splunk, Honeycomb and many more. These are well-established, well-documented products, and I won't dive into how to set them up here, but they all support ingesting logs from microservices with low coupling. In chapter 10, I'll assume that you have a Log microservice based on one of these products, and I'll show you how to send useful log messages to it.

Later in this chapter, we'll look at logging unhandled errors by adding handlers to MVCs pipeline.

7.1.2 Using correlation tokens

To find all log messages related to a particular action in the system, you can use *correlation tokens*. A correlation token is an identifier attached, for example, to a request from an end user when it comes into the system. The correlation token is passed along from microservice to microservice in any communication that stems from that end user request. Any time one of the microservices sends a log message to the Log microservice, the message should include the correlation token. The Log microservice should allow searching for log messages by correlation token. In figure 7.11, the API gateway would create and assign a correlation token to each incoming request; and the correlation token would then be passed with every microservice-to-microservice communication, including events and log messages. This is a well established pattern and can be implemented based on an open standard like Open Tracing which has the benefit that several of the off the shelf logging products mentioned above support and understand Open Tracing.

NOTE Chapter 10 discusses how to implement request logging and how to include correlation tokens in communications and log messages.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

7.1.3 Rolling forward vs. rolling back

When errors happen in production, you're faced with the question of how to fix them. In many traditional systems, if errors begin to occur shortly after deployment, the default response is to roll back to the previous version of the system. In a microservice system, the default can be different. As discussed in chapter 1, microservices lend themselves to continuous delivery. With continuous delivery, microservices are deployed frequently, and each deployment should be both fast and easy to perform. Furthermore, microservices are sufficiently small and simple that many bug fixes are also easy. This opens the possibility of *rolling forward* rather than rolling backward.

Why would you want to default to rolling forward? In some situations, rolling backward is complicated—in particular, when database changes are involved. When a new version that changes the database is deployed, the microservice begins to produce data that fits in the updated database. Once that data is in the database, it has to stay there, which may not be compatible with rolling back to an earlier version. In such a case, rolling forward may be easier.

7.1.4 Don't propagate failures

Sometimes, things happen around a microservice that may disturb its normal operation. We say that the microservice is *under stress* in such situations. There are many sources of stress, including the following:

- One of the machines in the cluster on which the microservice's data store runs has crashed.
- The microservice has lost network connectivity to one of its collaborators.
- The microservice is receiving unusually high amounts of traffic.
- One of its collaborators is down.

In all these situations, the microservice under stress can't continue to operate the way it normally does. That doesn't mean it's down, but it must cope with the situation.

When one microservice fails, its collaborators are put under stress. That means the collaborators are also at risk of failing. While the microservice is failing, its collaborators can't query, send commands, or poll events from the failing microservice. As illustrated in figure 7.12, if the collaborators fail, even more microservices become at risk of failing: the failure begins to propagate through the system of microservices. Such a situation can quickly escalate from one microservice failing to many microservices failing.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

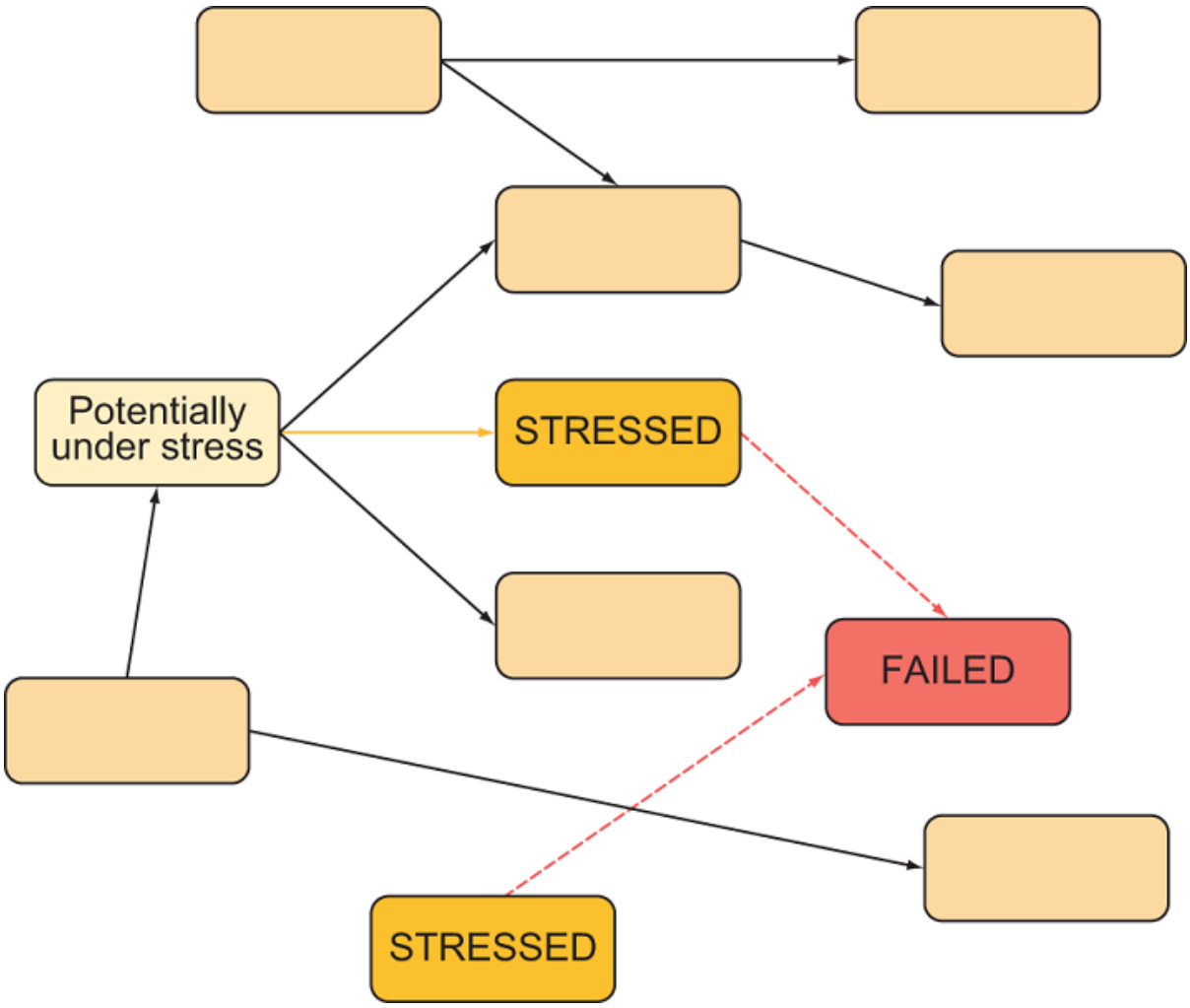

Figure 7.3 If the microservice marked FAILED is failing, so is communication with it. That means the microservices at the other end of those communications are under stress. If the stressed microservices fail due to the stress, the microservices communicating with them are put under stress. In that situation, the failure in one failed microservice can propagate to several other microservices.

Here are some examples of how you can stop failures from propagating:

- When one microservice tries to send a command to another microservice that happens to be failing at the time, that request will fail. If the sender fails too, you get the situation illustrated in figure 7.12, with failures propagating throughout the system. To stop the propagation, the sender can act as if the command succeeded, but actually store the command in a list of failed commands. The sending microservice can periodically go through the list of failed commands and try to send them again. This isn't possible in all situations, because the command may need to be handled immediately; but when this approach is feasible, it stops the failure in one microservice from propagating. This approach can be combined with a *circuit breaker*, which we'll talk about later in the chapter.
- When one microservice queries another that's failing, the caller can use a cached response. In chapter 6, you saw how to cache query responses and how to respect the cache header set by the microservice being queried. If the caller has a stale response in the cache, but a query for a fresh response fails, it may decide to use the stale response

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

anyway. Again, this isn't possible in all situations, but when it is, the failure won't propagate.

- An API gateway that's stressed because of high amounts of traffic from a certain client can throttle that client by not responding to more than a certain number of requests per second from the client. Note that the client may be sending an unusually high number of requests because it's failing internally. When throttled, the client will get a degraded experience but will still receive some responses. Without the throttling, the API gateway may become slow for all clients or fail completely. Moreover, because the API gateway collaborates with other microservices, handling all the incoming requests would push the stress of those requests onto other microservices, too. Again, throttling stops the failure in the client from propagating to other microservices.

As you can see from these examples, stopping failure propagation comes in many shapes and sizes. The important takeaway is the idea of building into your systems safeguards that are specifically designed to stop propagation of the kinds of failures you anticipate. How that's realized depends on the specifics of the systems you're building. Building in safeguards may take some effort, but it's often well worth the effort because of the robustness they give the system as a whole.

7.2 The client side's responsibility for robustness

When two microservices collaborate, there's a client and server, as shown in figure 7.13. The client is the microservice that sends out HTTP requests, and the server microservice handles them. The request shown in figure 7.13 happens to fail; it may fail because the server fails, or it may fail because it doesn't reach the server. Once the request has failed, the server can't do anything about it. The server can't send a response to a failed request—the request is already gone at that point. Responsibility for handling requests therefore must fall on the client. In other words, the client is responsible making the collaboration robust in the face of failing requests.

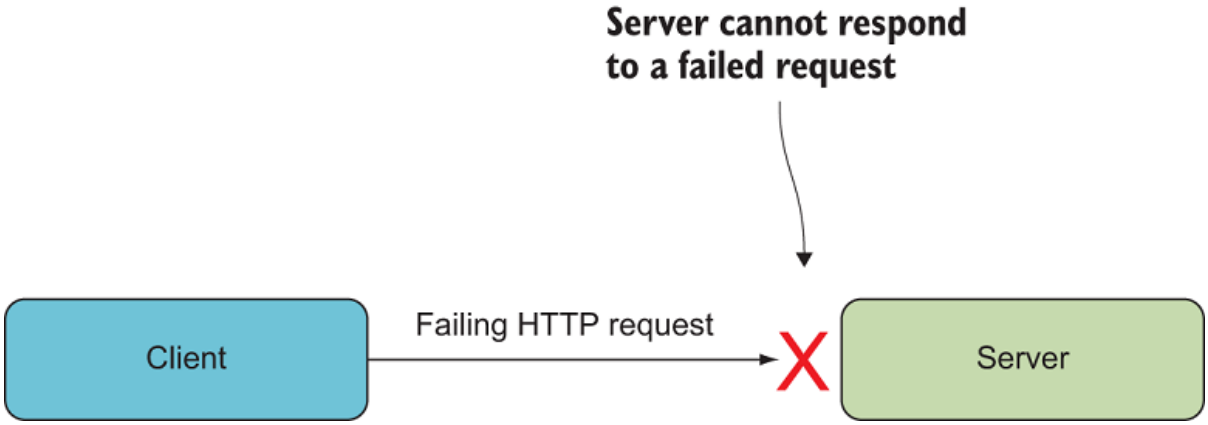

Figure 7.4 All collaborations between microservices have a client and a server. The client sends HTTP requests to the server. The client is responsible for handling failed requests.

When you look at an event-based collaboration, you see a degree of robustness in the face of failing requests built into the collaboration itself. Figure 7.14 shows an event subscriber in one microservice and an event feed in another microservice. As you saw in chapter 4, an event

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

subscriber polls the event feed at intervals for new events. That way of collaborating means that if a request for new events fails, the event subscriber will ask for the same events the next time it polls for events. The subscriber can catch up to events in the event feed even though some requests fail, and thus the subscriber is robust with regard to failing requests for events.

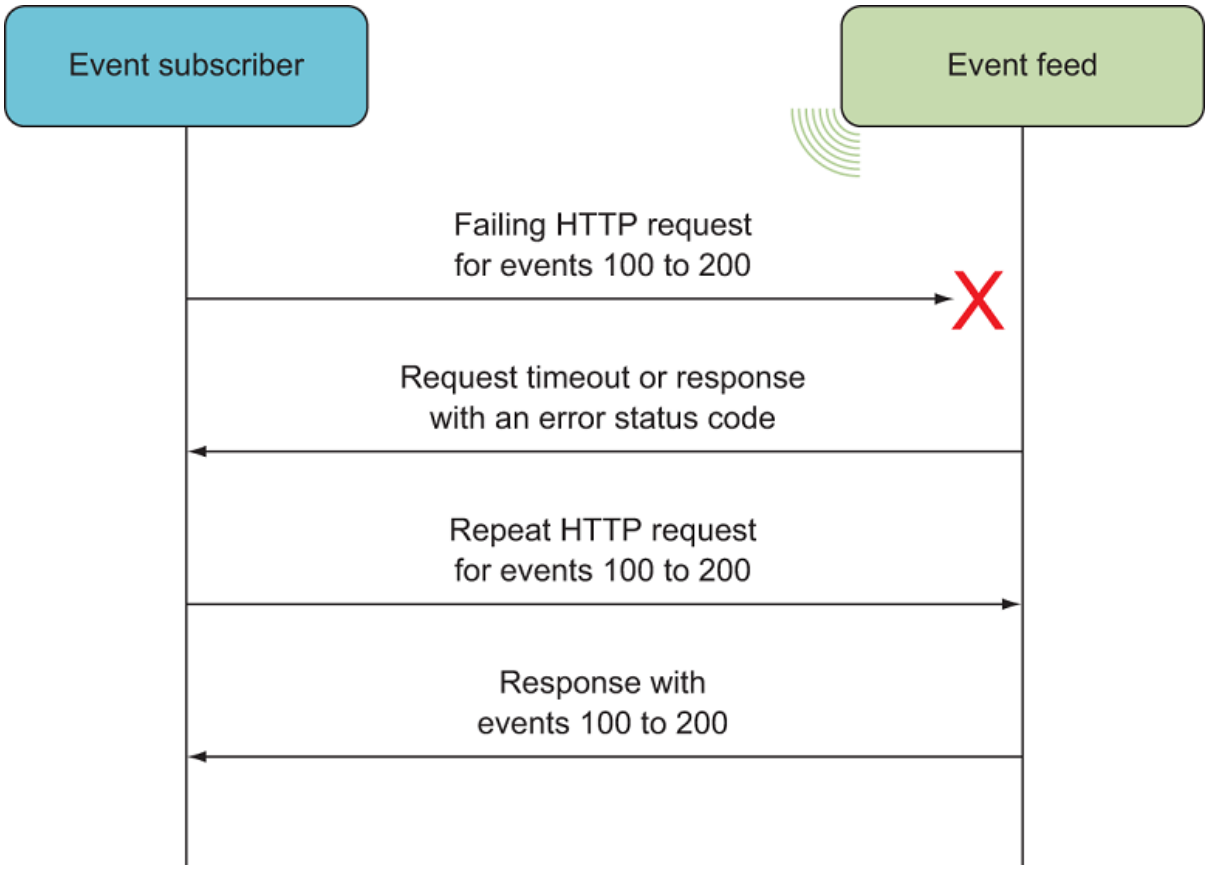

Figure 7.5 If a request for events from an event feed fails, the subscriber will request the same events the next time it polls for events.

You can see that with regard to command- and query-based collaboration, robustness doesn't come easily. The next two sections talk about patterns for building robustness into command- and query-based collaborations.

7.2.1 Robustness pattern: retry

The client in a command- or query-based collaboration may choose to try again when a request fails. If the reason for the failed request is transient, the next attempt may be successful. Transient failures are common, and the reasons for them include the following:

- Network congestion.
- The server microservice being deployed. Depending on how the microservice is deployed, there may be a short window when the microservice is unavailable or slow—for example, while a load balancer is switched over to a new version. Even if the server is slow only during deployment, requests may fail due to timeouts.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Retrying is a double-edged sword. If the reason for the failures isn't transient, retrying requests won't help. On the contrary, retrying indiscriminately puts stress on the server, because it's getting not only its usual number of requests but also the retries (see figure 7.15). This may not seem like a big deal, but imagine a system that's already under high load. During normal operation, the client sends many requests to the server. If requests start failing and the client retries all of them, the client ends up sending more and more requests to the server. If the reason for the failing requests is that the server is already having trouble keeping up with the number of requests it receives, sending even more requests certainly isn't going to help.

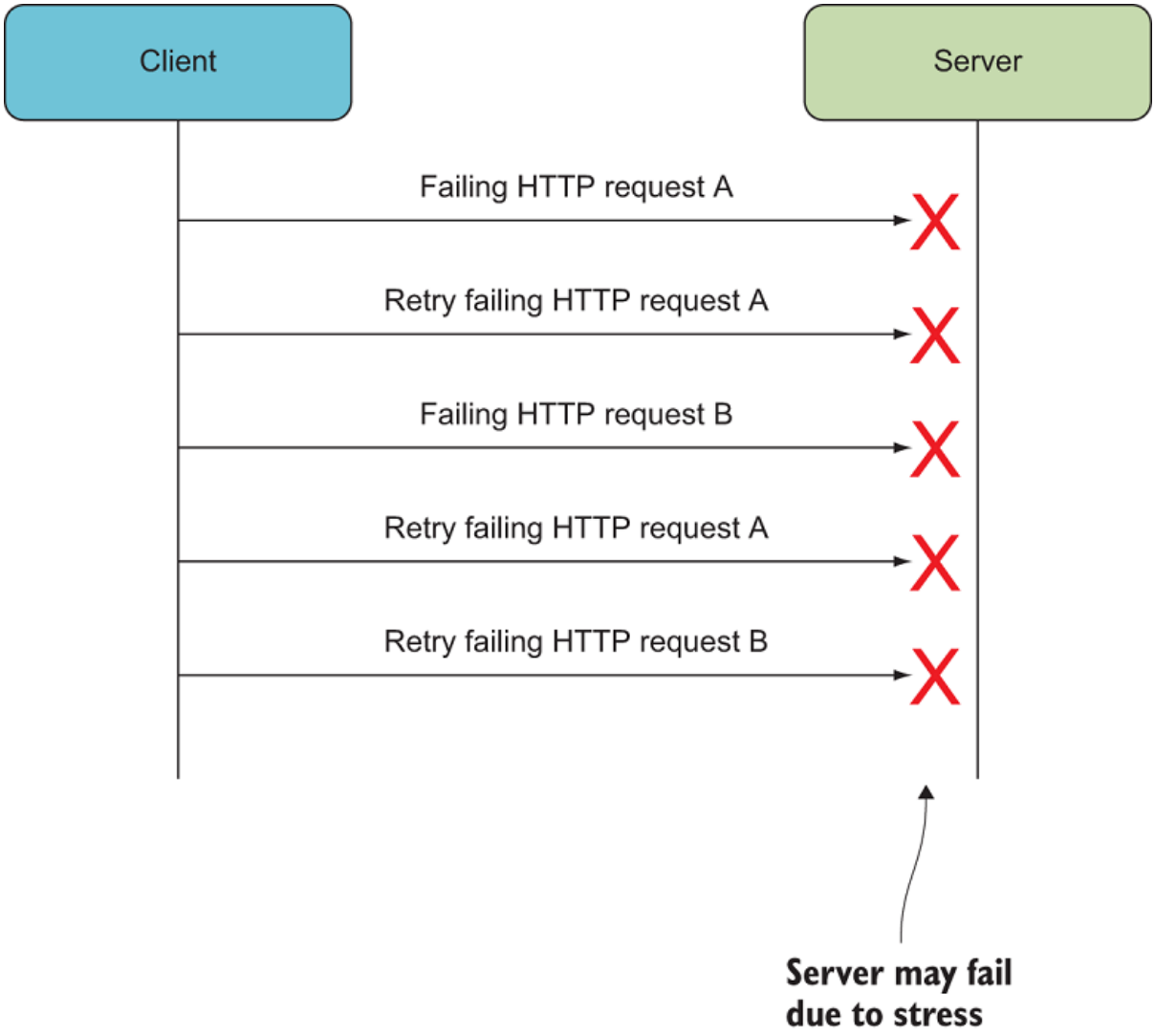

Figure 7.6 If a client keeps retrying a request that continues to fail at the server, the stress on the server will grow, and it may eventually fail completely.

Does this mean retrying is a bad pattern? No; it means you shouldn't continue retrying or retry too aggressively. The first thing to consider is how many times it makes sense to retry. If the request fails three times, is there any reason to believe it will succeed the fourth time? Second, you can use an exponential *backoff* between each retry. That is, instead of retrying after a constant amount of time (say, 100 ms), wait two or three times longer between each retry: maybe

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

100 ms before the first retry, 200 ms between the first and the second, and 400 ms between the second and third. These two simple additions mean the stress on the server builds up more slowly; you should always use them when you retry command or query requests. Later in this chapter, I'll show how the Polly library makes it easy to set up such retry strategies.

You may even want to consider making the interval between retries much longer. Instead of waiting a fraction of a second before retrying, you could wait a few minutes or even hours. Figure 7.16 shows a retry strategy in which the intervals are long and become exponentially longer. This type of retry doesn't place nearly as much stress on the server as the fast retries used in many software systems.

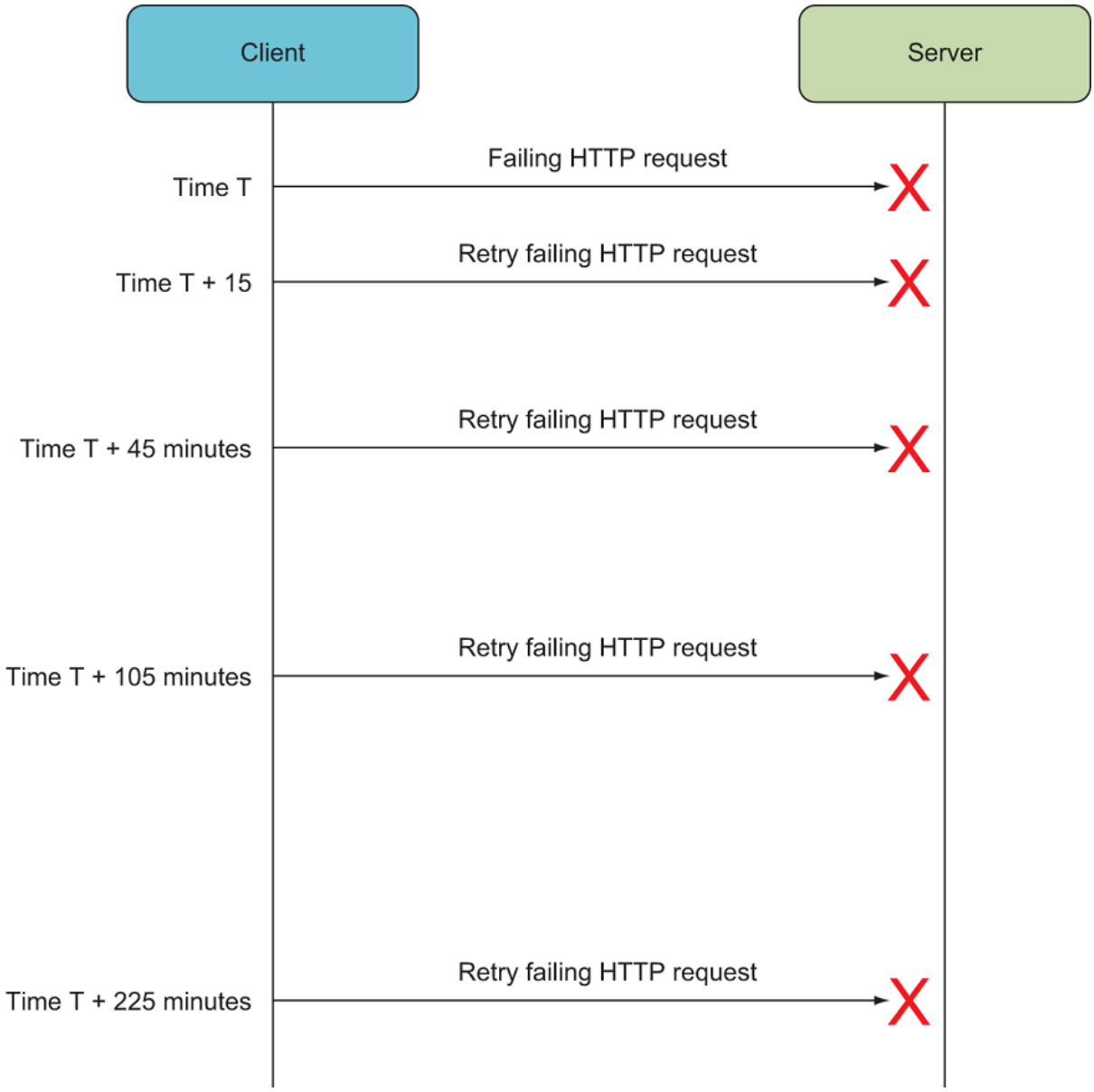

Figure 7.7 To avoid putting unnecessary stress on the server, the client can wait exponentially longer and longer between retries.

This approach clearly doesn't work for all situations. If a user is waiting for a response, it makes

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

no sense to retry an hour later, because the user will have given up long before, so the software should also give up and give a degraded response sooner. On the other hand, if the request is initiated based on something the system does on its own—for example, as part of handling an event—you can often wait a long time.

Next, we'll discuss another useful pattern for making collaborations robust: the circuit breaker pattern. Then we'll move on to code and implement both the fast-paced and slow-paced styles of retries.

7.2.2 Robustness pattern: circuit breaker

The circuit breaker pattern is a different take on dealing with failing requests. As you saw in the previous section, retrying failing requests can add to the problem by putting the server under stress; therefore, you must limit the number of retries. The circuit breaker pattern takes this line of thinking a step further: it assumes that if a number of different requests in a row fail, then the next request is also likely to fail.

Figure 7.17 illustrates this situation. The client has already made HTTP requests A, B, C, and D. Is it then likely that E will succeed? In many cases, no. The fact that a number of requests failed indicates that the problem isn't with the individual requests. Rather, it has to do with the communication—the client can't reach the server, the server is failing, or the client is sending bad requests. The issue with communication may be transient, but even so, request E often also fails.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

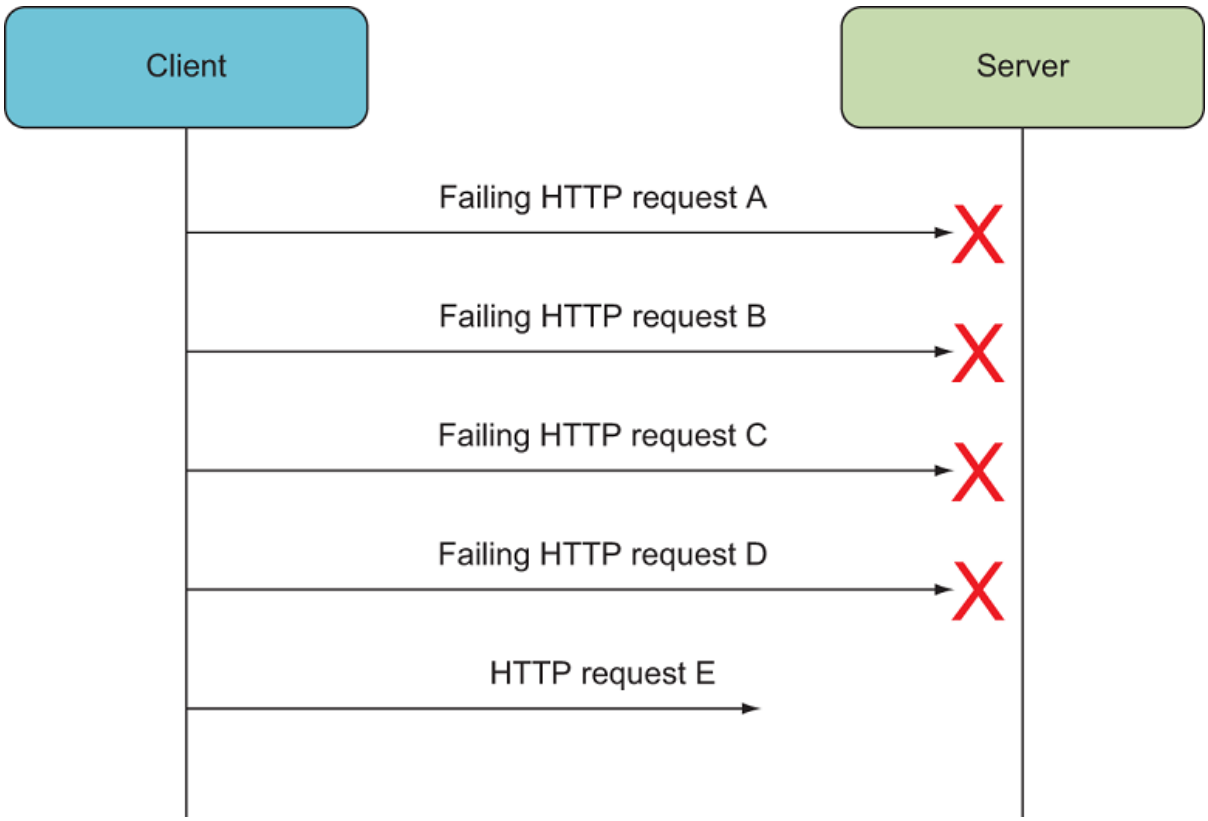

Figure 7.8 If several requests in a row have failed, is the next one likely to fail? In many cases, yes.

The circuit breaker pattern addresses this situation by not making request E at all, but instead assuming it will fail. Not making requests that are likely to fail alleviates stress on both the client and the server:

- The server receives fewer requests.
- The client doesn't have to wait for requests to fail, but rather assumes they will, meaning the client doesn't spend resources on waiting and can get its own work done more quickly.

A circuit breaker wraps HTTP requests in a state machine like the one shown in figure 7.18. When the microservice needs to make an HTTP request, it does so through the circuit breaker.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

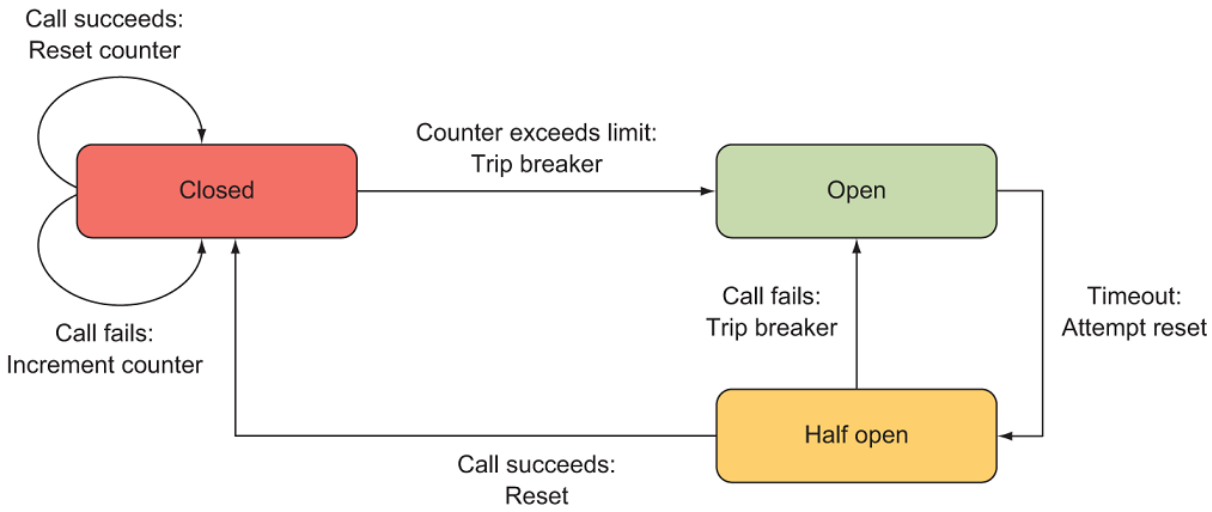

Figure 7.9 A circuit breaker is a state machine with three states. When the circuit breaker is closed, real HTTP requests are made. When the circuit breaker is open, no requests are made. A circuit breaker helps avoid making HTTP requests that are likely to fail.

The circuit breaker state machine starts in the closed state and works as follows:

- While the circuit breaker is in the *closed state*, it makes a real HTTP request when asked to. If the HTTP request fails, the circuit breaker increments a counter. If the request succeeds, the circuit breaker resets the counter to zero. When and if that counter exceeds a preset limit—say, five failed requests in a row—the circuit breaker goes to the open state.
- While the circuit breaker is in the *open state*, it doesn't make any HTTP requests. Instead, it errors immediately. The circuit breaker stays in the open state for a preset period—say, 30 seconds—and then goes to the half-open state.
- While the circuit breaker is in the *half-open state*, it makes an HTTP request the first time it's asked to. After that one HTTP request, it goes to the closed state if the request succeeded, or the open state if it failed.

The result of these simple rules is a state machine that stops making HTTP requests when they're likely to fail anyway. Later in this chapter, I'll show you how to use Polly to create circuit breakers around HTTP requests.

TIP Circuit breakers not only are useful for HTTP requests, but also can be used to add robustness around any other operation that can fail.

For the remainder of the chapter, we'll get down to the code level and see how to implement retry strategies and circuit breakers using Polly and general error handling using MVC's pipelines.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

7.3 Implementing robustness patterns

To see how to implement the retry and circuit breaker patterns discussed in the previous sections, let's turn our attention back to the point-of-sale system introduced in chapter 4 and zoom in on the collaborations around the Loyalty Program microservice. You identified these collaborations in chapter 5; figure 7.19 shows them again, annotated with the robustness strategies you'll implement in the following sections. We won't look at code for the robustness strategies of the Invoice and Log microservices, so they're grayed out.

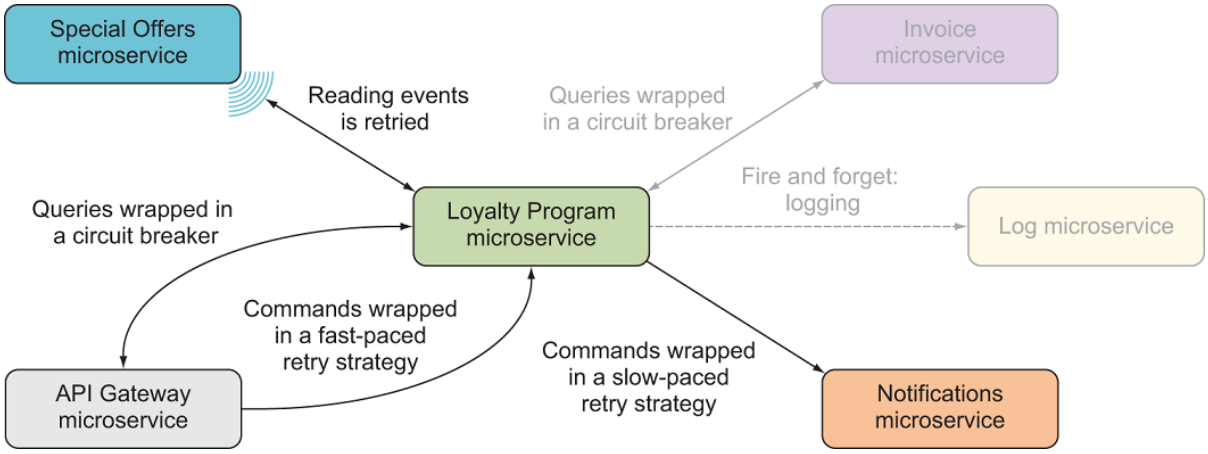

Figure 7.10 The Loyalty Program microservice collaborates with several other microservices. Each collaboration is annotated with a robustness strategy.

In the following sections, you'll do the following:

- Implement a fast-paced retry strategy in the API Gateway microservice for the commands it sends to the Loyalty Program microservice. The implementation is based on the Polly library.
- Implement a circuit breaker in the API Gateway microservice for the queries it makes to the Loyalty Program microservice. This implementation is also based on the Polly library.
- Implement a slow-paced retry strategy in the Loyalty Program microservice for the commands it sends to the Notifications microservice, based on the way the event subscription already works.
- Implement general exception handlers in the HTTP API in the Loyalty Program microservice using facilities in MVC.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

SIDEBAR **Polly**

Polly is a convenient library for creating and using error-handling strategies. Creating a strategy with Polly is done in a declarative way via a fluent API. Once created, a strategy can be applied to any `Func` or `Action`—which essentially means you can apply the strategy to any code you want. Furthermore the `HttpClient` class integrates with Polly to make applying error handling strategies to HTTP requests even easier.

The three basic steps to using Polly are as follows:

1. Decide which exceptions to handle, such as HttpException.
2. Decide which policy to use, such as a retry policy.
3. Apply the policy to a function.

The entry point to working with Polly is the `Policy` class:

```
var retryStrategy =
  Policy
    .Handle<HttpException>()                    ❶
    .Retry();                                   ❷

retryStrategy.Execute(() => DoHttpRequest());   ❸
```

❶ Step 1: Decide which exceptions to handle.

❷ Step 2: Decide which policy to use.

❸ Step 3: Use the strategy to wrap a function call.

Polly comes with a number of built-in policies, including variations of retry strategies and various circuit breaker strategies. Furthermore all strategies come in async as well as sync flavors.

We will leverage Polly and the integration with `HttpClient` to easily implement solid retry and circuit breaker strategies.

The API Gateway microservice consists of the components shown in figure 7.20. In the next two sections, we'll zoom in on `LoyaltyProgramClient`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

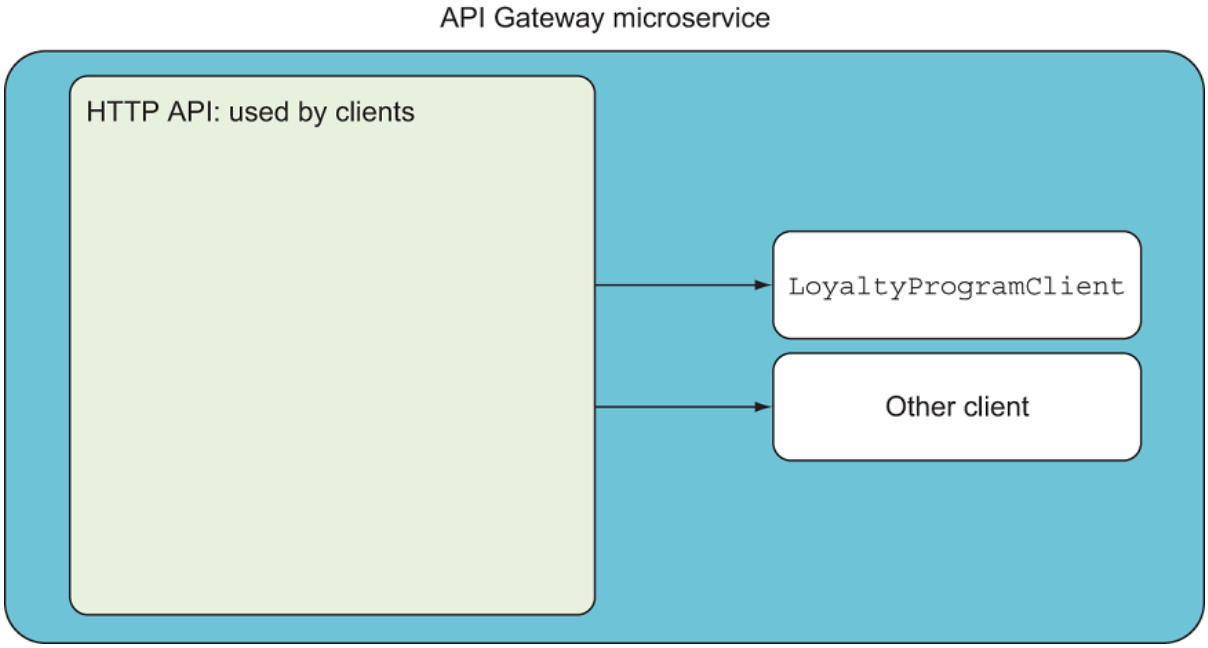

Figure 7.11 The API Gateway microservice consists of the same standard set of components you've seen several times already.

7.3.1 Implementing a fast-paced retry strategy with Polly

As shown in figure 7.19, the API Gateway microservice sends commands to the Loyalty Program microservice. We'll only look at adding a retry strategy to the register-user command here, because the code for adding a retry strategy to the update-user command is essentially the same.

First, you need to add the Polly NuGet package to the API Gateway microservice. The code that sends the commands to the Loyalty Program microservice is in `LoyaltyProgramClient`. You use Polly to set up a retry policy that uses an exponential backoff. Polly splits the setup of a policy from the execution of the policy: that is, Polly allows you to set up different policies—a retry policy, for instance—and then later execute a piece of code under the policy. In the case of a retry strategy, that means retrying the piece of code if it fails. The retry policy for the register-user command is set up as shown next.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 7.1 Polly retry policy

```
using System;
using Polly;

public class LoyaltyProgramClient
{
  private static readonly IAsyncPolicy<HttpResponseMessage>
    ExponentialRetryPolicy =
      Policy<HttpResponseMessage>                                    (1)
        .Handle<HttpRequestException>()                              (2)
        .OrTransientHttpStatusCode()                                 (3)
        .WaitAndRetryAsync(                                          (4)
          3,                                                         (5)
          attempt =>
            TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt))    (6)
        );
}
```

(1) Code executed under this policy should return an `HttpResponseMessage`

(2) Handles all http exceptions

(3) Handle timeouts and server erros (5XX status codes)

(4) Chooses an async policy because you'll use it with async code later

(5) Number of retries

(6) Time span to wait before the next retry

With the retry strategy set up, you can use it to wrap the call to the Loyalty Program microservice. We can chose to do this directly or through the integration between `HttpClient` and Polly. First let's look at using the retry policy directly `LoyaltyProgramClient`:

Listing 7.2 Using a Polly policy around an HTTP request

```
public async Task<HttpResponseMessage>
  RegisterUser(string name)
{
  var user = new {name, Settings = new { }};
  return await ExponentialRetryPolicy
    .ExecuteAsync(() =>                                              (1)
      this.httpClient.PostAsync("/users/", CreateBody(user)));       (2)
}

private static StringContent CreateBody(object user) => ...
```

(1) Executes an Action with the retry policy

(2) Makes the HTTP request

That is how easy it is to set up a retry policy with Polly and wrap a piece of code in the policy. We can achieve the same behavior in a slightly different way by configuring the `HttpClient` used in `LoyaltyProgramClient` with the retry policy. In this version `LoyaltyProgramClient` is left untouched. Instead we add the `exponentialRetryPolicy` to the `Startup` class where we use it to configure the `HttpClient` for `LoyaltyProgramClient` like this:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

```
public class Startup
{
  private static readonly IAsyncPolicy<HttpResponseMessage>
    ExponentialRetryPolicy =                                          ❶
      Policy<HttpResponseMessage>
        .Handle<HttpRequestException>()
        .OrTransientHttpStatusCode()
        .WaitAndRetryAsync(
          3,
          attempt =>
            TimeSpan.FromMilliseconds(100 * Math.Pow(2, attempt)));

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddHttpClient<LoyaltyProgramClient>()                    ❷
      .AddPolicyHandler(_ => ExponentialRetryPolicy)                  ❸
      .ConfigureHttpClient(c => c.BaseAddress = new Uri(host));       ❹
      ...
  }
  ...
}
```

❶ The retry policy as above

❷ Tell the service collection that `LoyaltyProgramClient` wrap an `HttpClient`

❸ Add retry policy to the `HttpClient`

❹ Add other configuration to the `HttpClient`

This code registers the `LoyaltyProgramClient` with the service collection, which allows ASP.NET Core to inject `LoyaltyProgramClient` into other types as a dependency. Furthermore the service collection is instructed that the `LoyaltyProgramClient` need an `HttpClient` which should apply the `ExponentialRetryPolicy` to all HTTP requests. This also achieves our goal of wrapping the user registration in a retry policy. Well, in fact, this does a little bit more that: Since we set up the `HttpClient` to wrap all Http request in the retry policy, the retry is also applied to queries made by `LoyaltyProgramClient`. We will fix this problem in the next section when we add a circuit breaker to the queries.

Whether you prefer to apply policy directly in the `LoyaltyProgramClient` or in configuration when adding the `LoyaltyProgramClient` to the service collection comes down to taste: Which version do you find simpler? Personally I don't have strong preference either way - as long as we remember to add robustness patterns to all collaboration.

Next, you'll use Polly to create a circuit breaker.

7.3.2 Implementing a circuit breaker with Polly

Now you'll add a circuit breaker to the API Gateway microservice's queries to the Loyalty Program microservice. This time, you'll use Polly's built-in support for circuit breaker policies.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 7.3 Polly circuit breaker policy

```
private static readonly IAsyncPolicy<HttpResponseMessage>
  CircuitBreakerPolicy =
    Policy<HttpResponseMessage>
      .Handle<HttpRequestException>()                    ❶
      .OrTransientHttpStatusCode()
      .CircuitBreakerAsync(5, TimeSpan.FromMinutes(1));  ❷
```

❶ Handles the same errors as the retry policy above]

❷ Sets the failure limit to 5 and the time-in-open-state limit to 3 minutes

Even though the circuit breaker pattern may seem more complicated than retrying, Polly makes it just as easy to set up a circuit breaker policy as a retry policy. Using a policy is the same no matter what the policy is. So if you prefer using the policy directly in `LoyaltyProgramClient` you can wrap the queries the same way we wrapped register-user commands above. If, on the other hand you prefer the `HttpClient` configuration approach we extend the configuration in `Startup` to choose the policy based on the type of HTTP request the `HttpClient` performs:

Listing 7.4 Wrapping a query in a circuit breaker

```
services.AddHttpClient<LoyaltyProgramClient>()
  .AddPolicyHandler(request =>
    request.Method == HttpMethod.Get                ❶
      ? CircuitBreakerPolicy                        ❷
      : ExponentialRetryPolicy)                     ❸
  .ConfigureHttpClient(c => c.BaseAddress = new Uri(host));
```

❶ Check which HTTP method. A GET is a query, everything else is a command

❷ Use circuit breaker for queries

❸ Use retry for commands

With these two policies in place, API Gateway takes responsibility for adding robustness to the collaboration with Loyalty Program. Next, we'll move on to the Loyalty Program microservice.

7.3.3 Implementing a slow-paced retry strategy

The Loyalty Program microservice subscribes to events from the Special Offers microservice. Based on the events, Loyalty Program sends commands to the Notifications microservice, asking it to notify users about new special offers. If sending a command to Notifications fails, you want to retry. Because sending out notifications isn't particularly time critical, you'll choose not to retry immediately; instead, you'll retry the next time the event subscriber would otherwise poll for new events. To do this, all you have to do is keep track of what the last successful event was.

Remember from chapter 5 that an event subscriber works by periodically waking up and polling the event feed for new events. On each such cycle, the next batch of events is read and handled.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

The next batch of events will begin one event after the last successfully handled event. This means all failed events are retried. It also means you may as well abort the rest of a batch as soon as one event fails—the rest will be retried later anyway. The code from chapter 5 handles one batch of events and was setup to be called by Kubernetes on a schedule. So we actually already implemented slow-paced retry in chapter 5. To emphasize how the slow-paced retry works let's re-iterate the code from chapter 5. First of there is a small program that handles one batch of events:

Listing 7.5 Single-event subscription cycle

```
namespace EventConsumer
{
  using System;
  using System.Net.Http;
  using System.Net.Http.Headers;
  using System.Threading.Tasks;

  class Program
  {
    static int start = 0;

    static async Task Main(string[] args)
    {
      start = GetStartIdFromDataStore();                    (1)
      var end = start + 100;
      var client = new HttpClient();
      client
        .DefaultRequestHeaders
        .Accept
        .Add(new MediaTypeWithQualityHeaderValue("application/json"));
      var resp =
        await client.GetAsync(                                          (2)
          new Uri($"http://special-offers/events?start={start}&end={end}"));
      var rawEvents = await resp.Content.ReadAsStringAsync();           (3)
      ProcessEvents(rawEvents);                                         (4)
      SaveStartIdToDataStore(start);                                    (5)
    }

    static int GetStartIdFromDataStore() { ... }
    static void SaveStartIdToDataStore(int start) { ... }
    static void ProcessEvents(string rawEvents) { ... }
  }
}
```

(1) Read the starting point of this batch from a database

(2) Send GET request to the event feed

(3) Read the response body. This a JSON array of events.

(4) Call method to process the events in this batch. ProcessEvents also updates the start variable.

(5) Save starting point of the next batch of events

This programs is called on a schedule and handles reading one batch of events. Once a batch has been read each event must be processed with code like this:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 7.6 Handling a batch of events

```
private async Task ProcessEvents(string rawEvents)
{
  var events = JsonConvert.DeserializeObject<IEnumerable<Event>>(rawEvents);
  foreach (var ev in events)
  {
    dynamic eventData = ev.Content;
    if (ShouldSendNotification(eventData))
      await SendNoitifcation(eventData);
    this.start = ev.SequenceNumber + 1;                                    (1)
  }
}

private bool ShouldSendNotification(dynamic eventData)
{
  // decide if notification should be sent based on business rules
}

private Task SendNotification(dynamic eventData)
{
  // use HttpClient to send command to notification microservice
}
```

(1) All events were assumed to be successfully handled, and "start" was updated for each one.

The specifics of the event handling code will differ with every specific situation - that's where the business logic is implemented.

With this in place one batch can be handle. Final piece is configuring Kubernetes to call the program on a schedule of once an hour which is done with this yaml that defines a CRON job in Kubernetes that call our event consumer every hour:

Listing 7.7 Kubernetes manifest for the LoyaltyProgram event consumer

```
---
apiVersion: batch/v1beta1                                  (1)
kind: CronJob                                              (2)
metadata:
  name: loyalty-program-consumer
spec:
  schedule: "*/1 * * * *"                                  (3)
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: loyalty-program
              image: microservicesindotnetregistry1.azurecr.io/loyalty-program:1.0.2
              imagePullPolicy: IfNotPresent
              env:
                - name: STARTUPDLL
                  value: "consumer/EventConsumer.dll"      (4)
          restartPolicy: Never
  concurrencyPolicy: Forbid                                (5)
```

(1) The Kubernetes API version needed to specify a cron job

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

❷ Indicate that this is a cron job

❸ Define the schedule for this job.

❹ Point to the event consumer dll

❺ Make sure only one copy at a time of the event consumers runs

As you can see, we already implemented slow-paced retry, because that is how our event consumers work.

7.3.4 Logging all unhandled exceptions

Finally, we'll turn our attention to the HTTP API of the Loyalty Program microservice. As stated earlier, you want to keep good logs of everything that goes wrong in the system. That means you should log any unhandled exceptions thrown in the controllers in the microservices. In fact, the Loyalty Program microservice already does this. Part of the default ASP.NET Core pipeline is to catch all - otherwise - unhandled exceptions and log them out to standard out. To see this in action you can add a new endpoint to the `UserController` that simply throws a `NotImplementedException`:

Listing 7.8 An endpoint that always throws an exception

```
[HttpGet("fail")]
public IActionResult Fail() => throw new NotImplementedException();
```

Then run the Loyalty Program microservice with `dotnet run` and call the new endpoint at `https://localhost:5001/users/fail`. The output in the console will include an error log caused by the `NotImplementedException` similar to this:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Listing 7.9 Errors are logged to standard out:

```
dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\chors\Documents\horsdal3\code\Chapter07\LoyaltyProgram\LoyaltyProgram
fail: Microsoft.AspNetCore.Server.Kestrel[13]
  ❶
      Connection id "0HM2KQ4LI56GR", Request id "0HM2KQ4LI56GR:00000001": An unhandled exception was thr
System.NotImplementedException: The method or operation is not implemented.
   at LoyaltyProgram.Users.UsersController.Fail() in C:\Users\chors\Documents\horsdal3\code\Chapter07\Lo
   at lambda_method(Closure , Object , Object[] )
   at Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(Object target, Object[] parameters)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.SyncActionResultExecutor.Execute(IAct
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeActionMethodAsync()
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Ob
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeNextActionFilterAsync()
--- End of stack trace from previous location where exception was thrown ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Rethrow(ActionExecutedContextSeale
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Ob
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync()
--- End of stack trace from previous location where exception was thrown ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeFilterPipelineAsync>g__Awaited|19_0
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Awaited|17_0(ResourceInvok
   at Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g__AwaitRequestTask|6_0(Endpoint endpoint,
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.ProcessRequests[TContext](IHtt
```

❶ Start of the error log

In chapter 10 we will return to logging and see how to configure our logging and how to make it even more useful, but for now we are happy with having all exceptions logged.

7.3.5 Deploying to Kubernetes

We have implemented robustness patterns in all the collaborations around the Loyalty Program microservice, but we haven't changed the collaboration styles - queries are still queries, commands are still commands, and events are still events. Moreover the dependencies between the microservices are still the same: The Loyalty Program microservice still subscribes to events from the Special Offers microsrevice and still sends commands to the Notifications microservice. This means that the deployment to Kubernetes does not have to change and is therefore exactly the same as we saw in chapter 5.

This concludes the implementation of robustness measures in collaborations around the Loyalty Program microservice. With these fairly simple measures in place, the collaborations are likely to be a good deal more robust under production load.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

7.4 Summary

- Due to the amount of communication between microservices, you must expect some communication to fail. It’s vital for the robustness of the system that your microservices handle such failures gracefully.
- You should design robustness into your microservices such that failures don’t propagate through the system and eventually become errors.
- The client side of a collaboration is responsible for making communication robust in the face of failures.
- You should have good logs that are easy to access and search through when you need to investigate production problems. A central Log microservice should receive all log messages and provide access to them.
- The most important strategies for making communications robust are the retry and circuit breaker patterns.
- Polly makes it easy to set up and use retry policies as well as circuit breakers.
- By default ASP.NET Core logs errors to standard out. The off shelf logging products support gathering up these logs from all the microservices and centralizing them.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Notes

1. Nicole Forsgren, Gene Kim, Jez Humble. *Accerate*. IT Revolution Press: 2018

2. Nicole Forgresn, Dustin Smith, Jez Humble, Jessie Frazelle: Accelerate: Stateof DevOps 2018. https://inthecloud.withgoogle.com/state-of-devops-18/dl-cd.html

3. For further discussion of what characterizes microservices, I recommend this article on the subject: Martin Fowler and James Lewis, "Microservices: A Definition of This New Architectural Term," March 25, 2014, http://martinfowler.com/articles/microservices.html.

4. Robert C. Martin, "SRP: The Single Responsibility Principle," http://http://mng.bz/zQyz.

5. Robert C. Martin, "The Single Responsibility Principle," May 8, 2014, http://mng.bz/RZgU.

6. Nicole Forsgren, Gene Kim, Jez Humble. *Accerate*. IT Revolution Press: 2018

7. Some microservices advocates argue that the correct way to arrive at microservices is to apply the Strangler pattern repeatedly to different subcomponents of the monolith. See Martin Fowler, "MonolithFirst," June 3, 2015, http://martinfowler.com/bliki/MonolithFirst.html.

8. techempower benchmarks

9. Martin Fowler: "Inversion of Control Containers and the Dependency Injection pattern", https://martinfowler.com/articles/injection.html

10. Melvin Conway, “How Do Committees Invent?” Datamation Magazine (April 1968).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

Index Terms

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/microservices-in-net-core-second-edition/discussion