Microservices in .NET Core

with examples in Nancy

Christian Horsdal Gammelgaard

SAMPLE CHAPTER

MANNING

Microservices in .NET Core

by Christian Horsdal Gammelgaard

Chapter 3

brief contents

Part 2

Building microservices

In this part of the book, you'll learn how to design and code a microservice. The assorted diverse topics all go into designing and coding good, maintainable, reliable microservices:

- Chapter 3 explains how to slice and dice a system into a cohesive set of microservices.
- Chapter 4 shows you how microservices can collaborate to provide functionality for end users. You'll also be introduced to three categories of collaboration and when to use each of them.
- Chapter 5 explores where the data goes in a microservice system and which microservices should take responsibility for which data.
- Chapter 6 teaches you some simple techniques to make a microservice system more robust than it would otherwise be. Using these techniques, you can create a system that keeps running in the face of network failures and individual microservice crashes.
- Chapter 7 turns to testing. You'll learn how to create an effective automated test suite for a microservice system, all the way from broad system-level tests to narrowly focused unit tests.

By the end of part 2, you'll know how to design microservices and how to use .NET Core and Nancy to code them.

3 Identifying and scoping microservices

This chapter covers

- Scoping microservices for business capability
- Scoping microservices to support technical capabilities
- 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 too long. 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 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.

The primary driver in identifying and scoping microservices is business capabilities; the secondary driver is supporting technical capabilities. Following these two

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.

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.

3.1 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.

3.1.1 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 and thus only partly automates it. In both cases, the scope of the microservice is the business capability.

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.

(continued)

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.

3.1.2 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*:[1]

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

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

[1] Melvin Conway, "How Do Committees Invent?" *Datamation Magazine* (April 1968).

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.

3.1.3 Example: point-of-sale system

The example we'll explore in this chapter is a point-of-sale system, illustrated in figure 3.1. 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.

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 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.

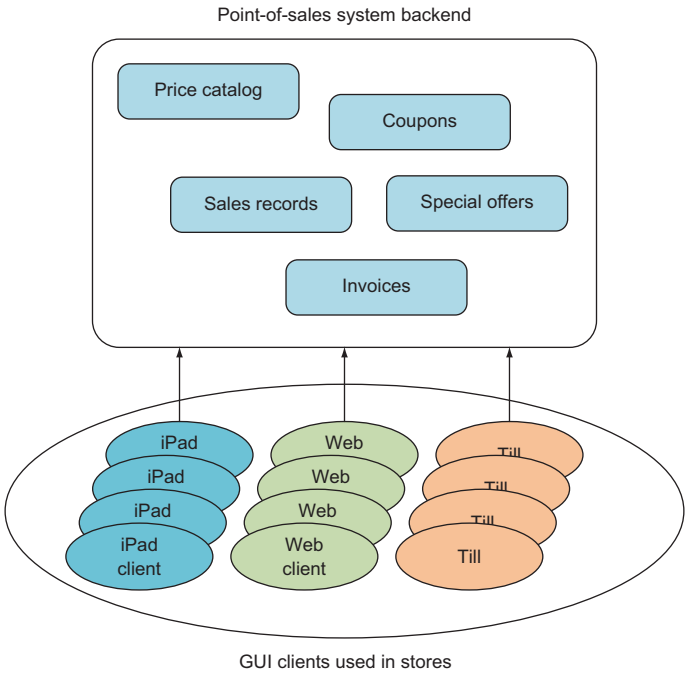

Figure 3.1 A point-of-sale system for a large chain of stores, consisting of a backend that implements all the business capabilities in the system and thin GUI clients used by cashiers in the stores. Microservices in the backend implement the business capabilities.

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
- 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
- Invoice
- Payment

Figure 3.2 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.

I find creating this kind of map to be enlightening, because it forces me to think explicitly about how each function is attained and also what each business capability

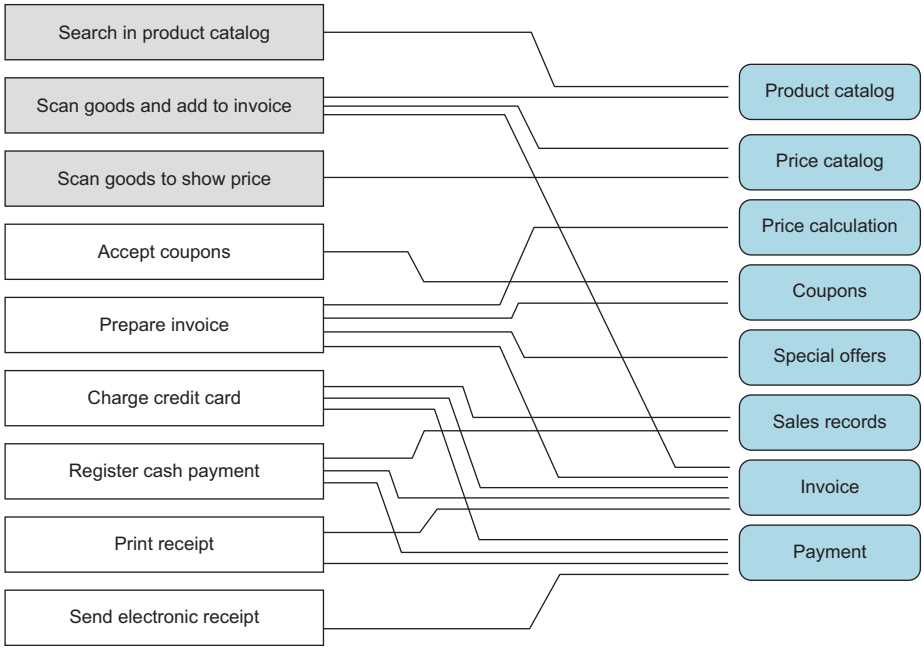

Figure 3.2 The functions on the left depend on the business capabilities on the right. Each arrow indicates a dependency between a function and a capability.

must do. 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.

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 3.3. Each process delivers part of the business capability.

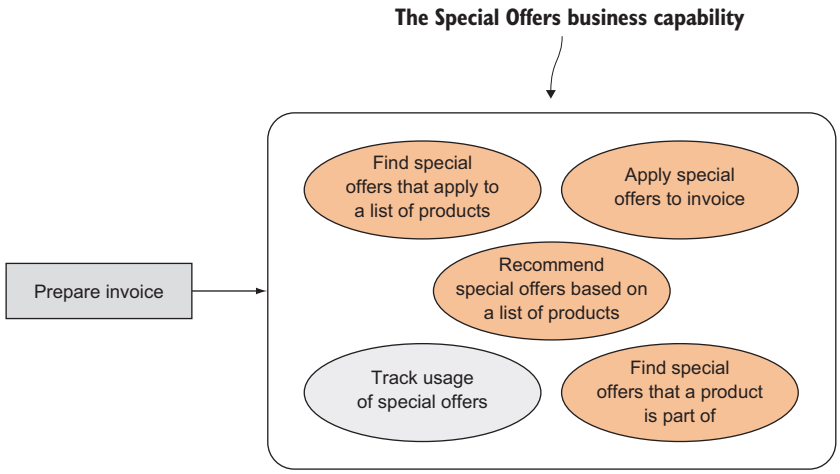

Figure 3.3 The Special Offers business capability includes a number of different processes.

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 3.4.

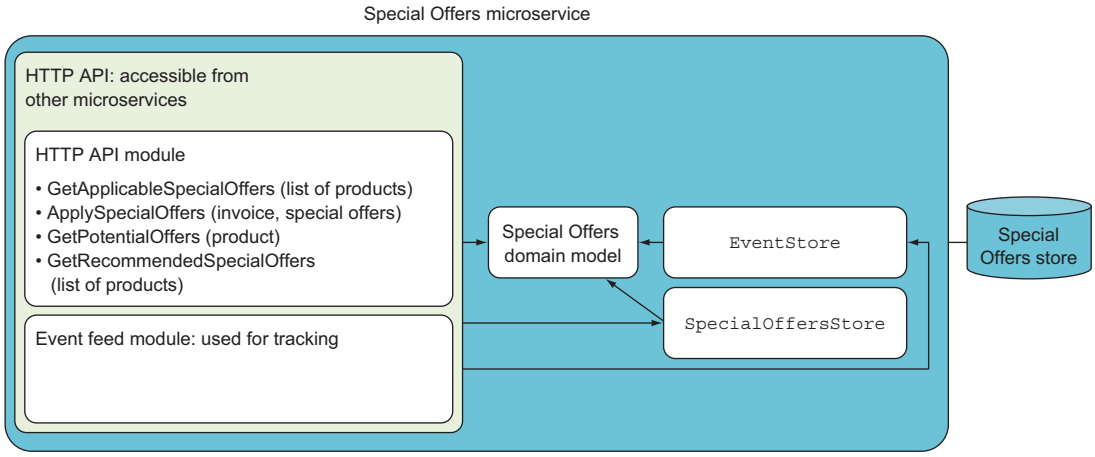

Figure 3.4 The processes in the Special Offers business capability are reflected in the implementation of the Special Offers microservice. The processes are exposed to other microservices through the microservice's HTTP API.

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 3.5. This is no coincidence. These are the components 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.

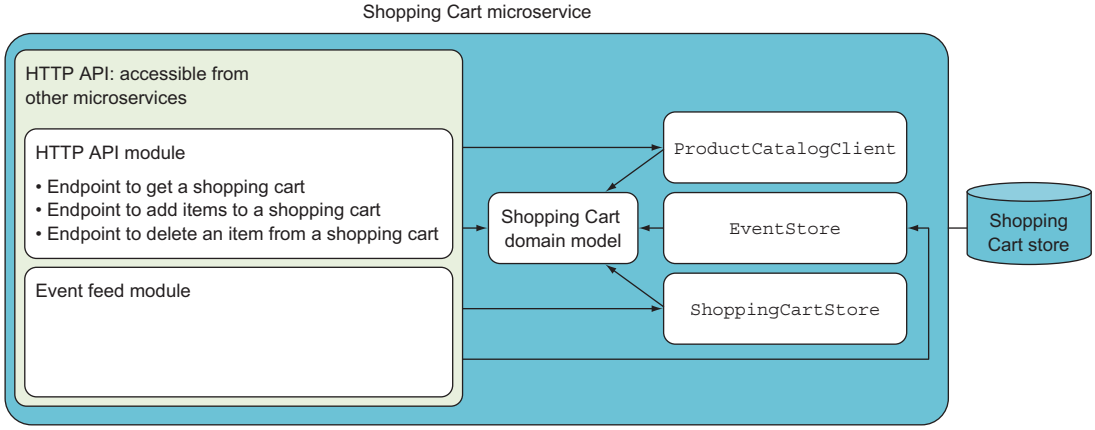

Figure 3.5 The components of the Shopping Cart microservice from chapter 2 are similar to the components of the Special Offers microservice.

3.2 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.

3.2.1 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.

3.2.2 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 3.6. The integration can be handled in a separate microservice.

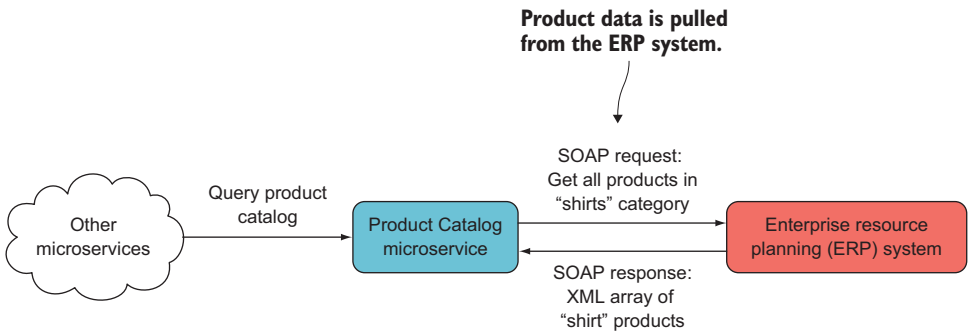

Figure 3.6 Product data flows from the ERP system to the Product Catalog microservice. The protocol used to get product information from the ERP system is defined by the ERP system. It could expose a SOAP web service for fetching the information, or it might export product information to a proprietary 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.

You'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 3.7. You'll do this for two reasons:

- By moving the technical complexities of the integration to a separate microservice, you 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, you 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.

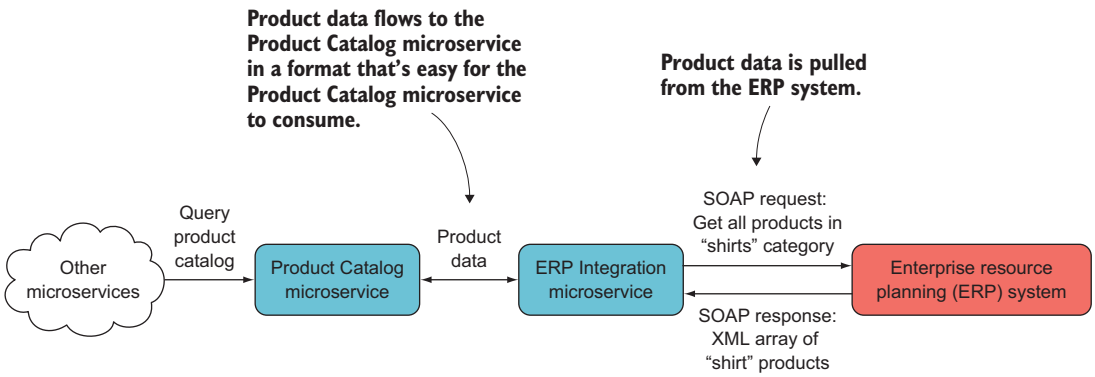

Figure 3.7 The ERP Integration microservice supports the Product Catalog microservice by handling the integration with the ERP system. It translates between the way the ERP system exposes product data and the way the Product Catalog microservice consumes it.

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 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. You 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 3.8 shows this microservice, which is responsible for notifying registered customers every time a new special offer is available.

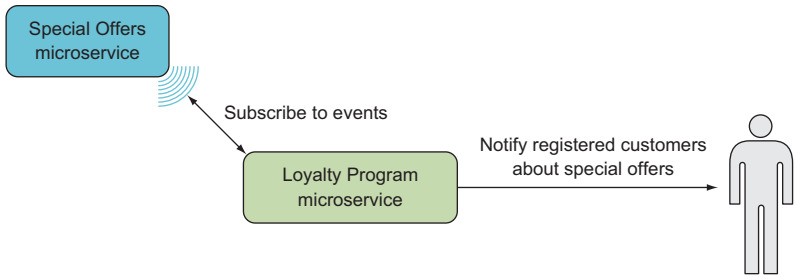

Figure 3.8 The Loyalty Program microservice subscribes to events from the Special Offers microservice and notifies registered customers when new offers are available.

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, you'll introduce a supporting technical microservice for each notification type. This is shown in figure 3.9.

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

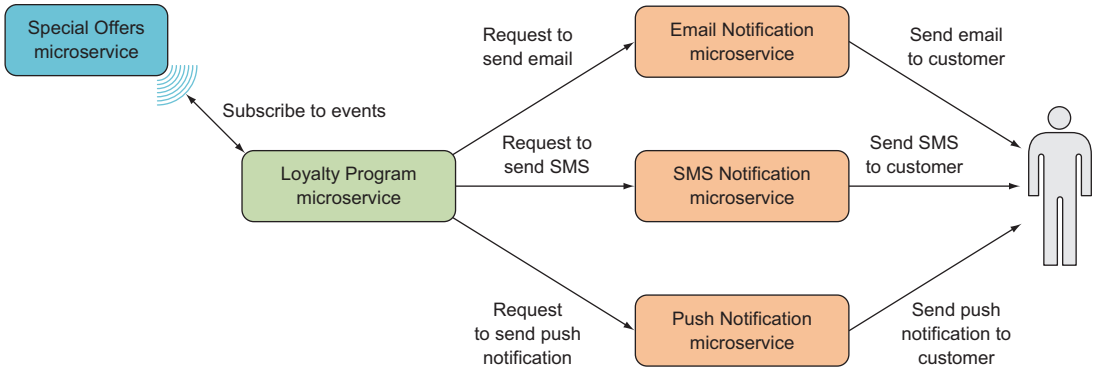

Figure 3.9 To avoid bogging down the Loyalty Program microservice in technical details for handling each type of notification, you'll introduce three supporting technical microservices, one for each type of notification.

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 you 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 3.10 and is responsible for choosing which type of notification to use each time a customer needs to be notified. This isn't really a business capability, although it's less technical than dealing with sending SMSs. I consider the Notifications microservice a supporting technical microservice rather than one implementing a business capability.

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

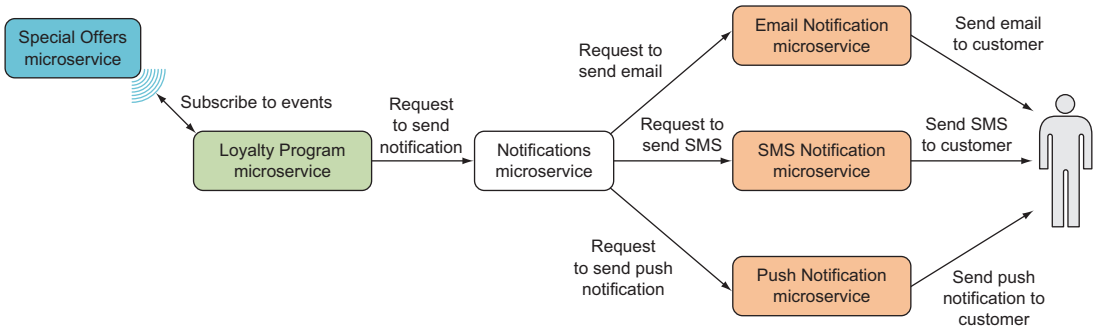

Figure 3.10 To remove more complexity from the Loyalty Program microservice, you'll introduce a Notifications microservice that's responsible for choosing a type of notification based on customer preferences. Introducing this microservice has the added benefit of making notifications easier to use from other microservices.

of that business capability can also take advantage of the Notifications microservice. Part of the motivation for moving this supporting technical capability to separate microservices is that you can reuse the implementation.

3.2.3 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 technical capability that several microservices need, and you turn that into a microservice 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 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 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.

3.3 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, and you also have to judge the complexity of supporting technical

capabilities correctly. 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.

3.3.1 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 3.11. 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 3.11.

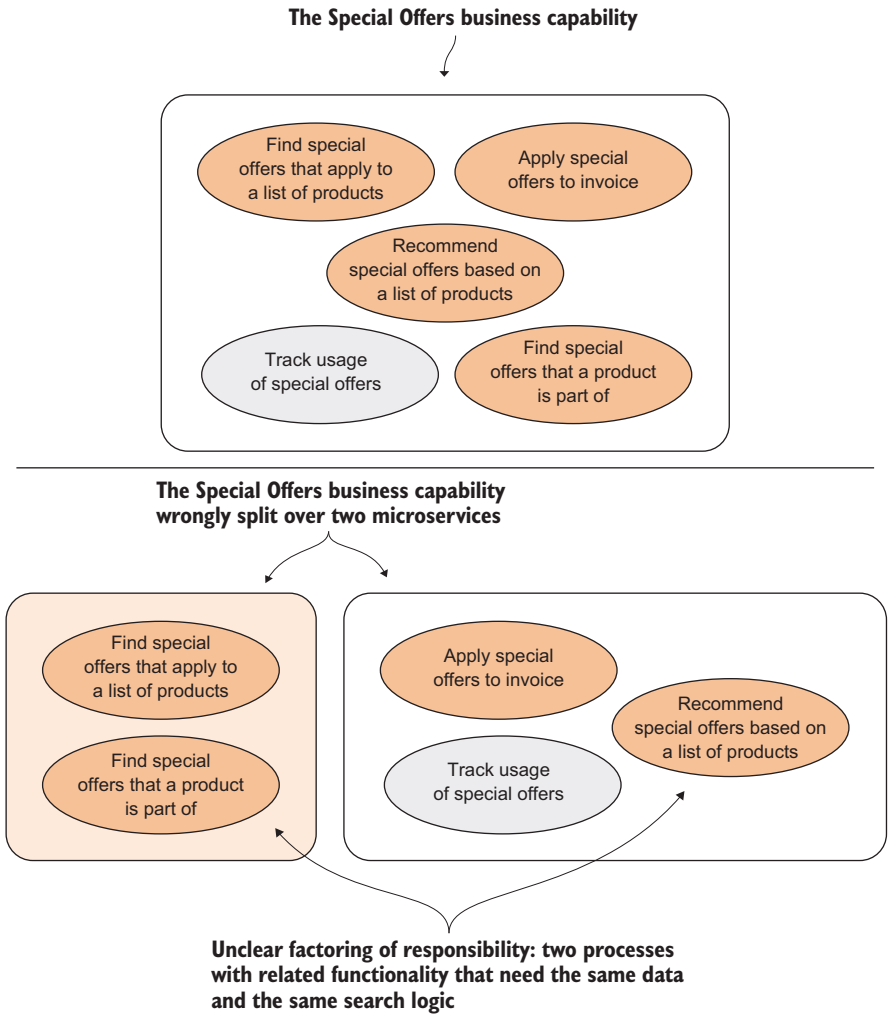

Figure 3.11 If you make the scope of a microservice too small, you'll find that a single business capability becomes split over several highly coupled parts.

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 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 3.12.

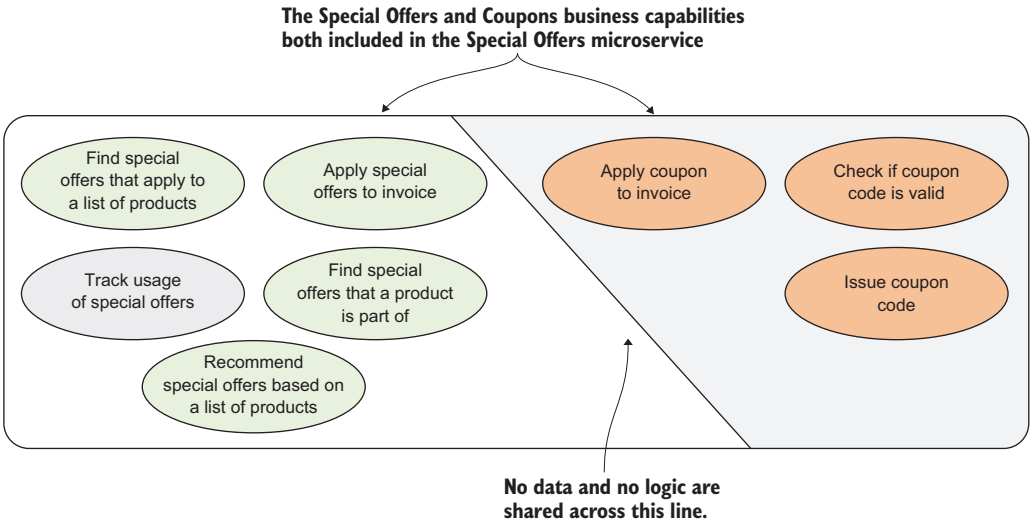

Figure 3.12 If you choose to err on the side of bigger scopes, you might decide to include the handling of coupons in the Special Offers business capability.

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, so 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 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.

3.3.2 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 3.13 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.

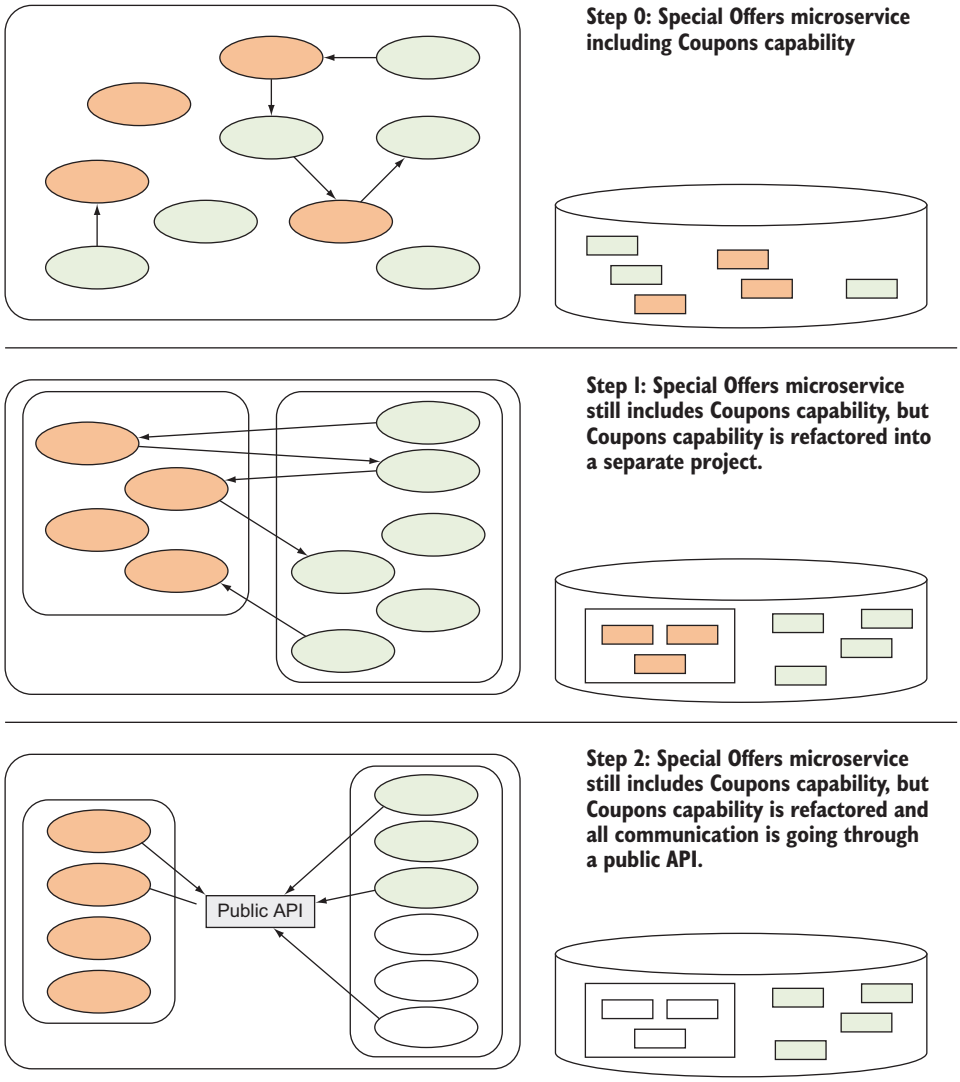

Figure 3.13 Preparing to carve out a new microservice by refactoring: first move everything belonging to the new microservice into its own project, and then make all communication go through a public API similar to the one the new microservice will end up having.

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.

3.3.3 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.

3.4 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.

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.

3.4.1 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.

3.4.2 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 3.2.2), it's fine to have one supporting technical microservice use others behind the scenes.

3.5 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.
- 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.

Microservices in .NET Core

Christian Horsdal Gammelgaard

Free eBook
SEE INSERT

Microservice applications are built by connecting single-capability, autonomous components that communicate via APIs. These systems can be challenging to develop because they demand clearly defined interfaces and reliable infrastructure. Fortunately for .NET developers, OWIN (the Open Web Interface for .NET), and the Nancy web framework help minimize plumbing code and simplify the task of building microservice-based applications.

Microservices in .NET Core provides a complete guide to building microservice applications. After a crystal-clear introduction to the microservices architectural style, the book will teach you practical development skills in that style, using OWIN and Nancy. You'll design and build individual services in C# and learn how to compose them into a simple but functional application back end. Along the way, you'll address production and operations concerns like monitoring, logging, and security.

What's Inside

- Design robust and ops-friendly services
- Build HTTP APIs with Nancy
- Expose events via feeds with Nancy
- Use OWIN middleware for plumbing

This book is written for C# developers. No previous experience with microservices required.

Christian Horsdal Gammelgaard is a Nancy committer and a Microsoft MVP.

"A definite must-read for anyone who works in C#/.NET regularly."
—Nick McGinness, Direct Supply

"Elegant and convincing. Developers will rethink their application architecture."
—James McGinn
Bull Valley Software

"Brings together two modern technologies and delves deeply into the code."
—Andy Kirsch
Concur Technologies

"An extremely approachable book that tackles a complex topic."
—Shahid Iqbal
Head For Cloud

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/microservices-in-net-core

ISBN-13: 978-1-61729-337-5
ISBN-10: 1-61729-337-7
54999
9 781617 293375

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]