1
Splitting the System

In the previous chapter, we got some insights into the concept of **Bounded Context**. Together with **Ubiquitous Language**, it forms the essence of **Domain-Driven Design** (**DDD**). As Vaughn Vernon often puts it, DDD is *developing Ubiquitous Language within Bounded Context*. At the same time, both of those concepts are probably the least understood and least paid-attention-to among people who ever tried to use DDD in their work. The previous chapter aimed to give you an overall idea of the concept itself, and I stress the importance of the language and the context in which the language applies throughout the book.

Now, let's see how the notion of Bounded Context can be put in practice.

In this chapter, we will cover the following topics:

- When it is a good time to think about splitting the system (and when it is not)
- What exactly you will be looking at when you have decided to separate your system into smaller parts
- How the system can be split by keeping the team sane and productive

When, what, and how?

I assume that after reading `Chapter 12`, *Bounded Context*, you would like to have a look at some systems that you are working on with a critical eye and try to understand whether you deal with a single system or multiple systems in disguise. You might already have a system that is composed of various components, but this doesn't automatically mean that those components are aligned with true business capabilities, which they should be. Let's start by looking into two or three questions that usually arise when we start thinking about splitting the system into pieces—when to split, what to break, and how to divide.

When to split

Although we keep hearing stories about successful, innovative companies starting with or moving to distributed systems and microservice architectures, the fact of life is that monolithic systems dominate the world of software. I already mentioned before that the **big ball of mud** is unarguably the most popular architectural style and explained the reasons behind it. But let's not forget that many monolithic systems are actually quite successful. After a few years of hype and fuzz around microservices, I keep seeing articles and blog posts with stories of moving back to a monolithic system after spending a tremendous amount of time trying to get a distributed system working.

Although the microservices topic is more suitable for the *How to split* section, I wanted to touch upon it, because, very often, the decision of using microservices comes quite early during the design phase, when we plan to build a new system or improve an existing system.

In my experience, the decision to split the system and align its components with business capabilities, effectively embracing the idea of Bounded Contexts, needs to appear in such discussions as early as possible, but not too early. I will try to explain now what I mean since the previous sentence might be a bit puzzling.

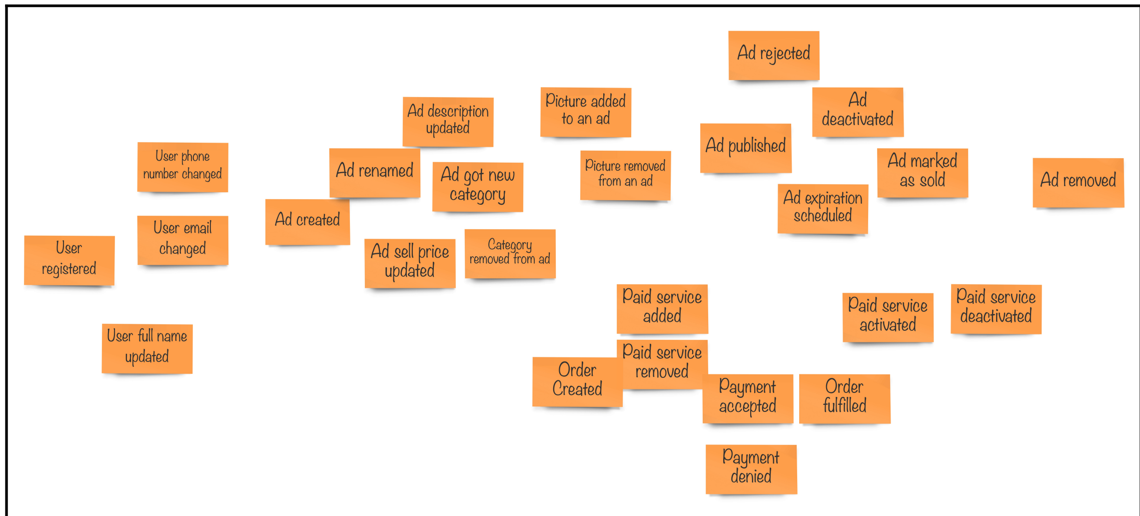
When we are designing something new, it makes sense to consider the new system as a set of components from the start, but only if the system we are going to build will be addressing a few business cases. It might be that we are starting to develop a prototype so our first phase would be to measure the interest of potential customers in our solution. We might decide to build a landing page or a straightforward mobile app that will talk to Firebase without even having a backend API. Such a design makes perfect sense in a discovery phase. Would I suggest splitting the system at that time? Definitely not. If you don't even clearly know what to build, you can't possibly imagine how you are going to make it. Remember that a deep understanding of the domain and a build-up of the Ubiquitous Language is a must before we can even talk about Bounded Contexts and such. How can we identify the context if we don't even speak the language? So, if you have a team of three or five working together in one room and building something at a very early stage, don't think about it too much. When you got to the point when your team grows, more business capabilities become requirements, and you start feeling that the meaning of words become somewhat unclear in different parts of the code or discussions with your business experts and customers—then you should consider taking a break and think about the language, and context, and how your system landscape follows them.

Let's consider a scenario for when you are working with an existing system. It might be an old, beasty monolith from the late 1990s, with a cumbersome VB.NET code mixed with T-SQL. It might also be a distributed system with tens or hundreds of microservices, built according to *resumé-driven design*, when you see pieces of different tech that were cool one, three, or five years ago, scattered across the network and communicating with each other with REST APIs (or maybe even gRPC; doesn't matter really). In such an environment, if you get a chance to build a new feature or refactor something that exists, but brings a lot of pain to customers and to the business itself, you definitely need to consider looking at the system and try to understand whether you can apply some principles discussed in this book.

Finally, when you are working in a company that has several development teams, which might be co-located or scattered across multiple offices around the globe, you definitely must look at splitting the system that you are building. It might very well be that you are fortunate, and, in your company, all those teams feel just enough to need to meet once a year at the big corporate event. All other communication is limited by infrequent alignment meetings and knowledge-sharing sessions. But, if you observe continuous communication between teams hanging tasks that are marked **blocked** with a red spot on them, where the reason for the block is another team not being able to deliver something in time, consider it as a classic case of incorrect boundaries and remember Conway's Law. Teams having trouble with coordination need be helped by realigning the scope of their work and removing dependencies by bringing a certain degree of isolation (in a good sense) and autonomy. Only when this is done can they become productive. These issues are never solved by bringing in more teams, and more people to the existing teams; it just makes the problem worse. Organize teams in a way that they have their own Bounded Context (or two, or three) to work on as a separate piece of the system that communicates with the world using published contracts only; this is the only way out of the coordination madness.

Emerging boundaries

Since we are discussing the *when* question, I have an example where boundaries emerge quite early in the product design phase. You might remember that I mentioned this possibility as a possible outcome of the EventStorming session in `Chapter 3`, *EventStorming*. If we look back at one of the design sessions for our sample domain, we can see, more or less, the full picture that came out of the workshop:



EventStorming workshop outcome

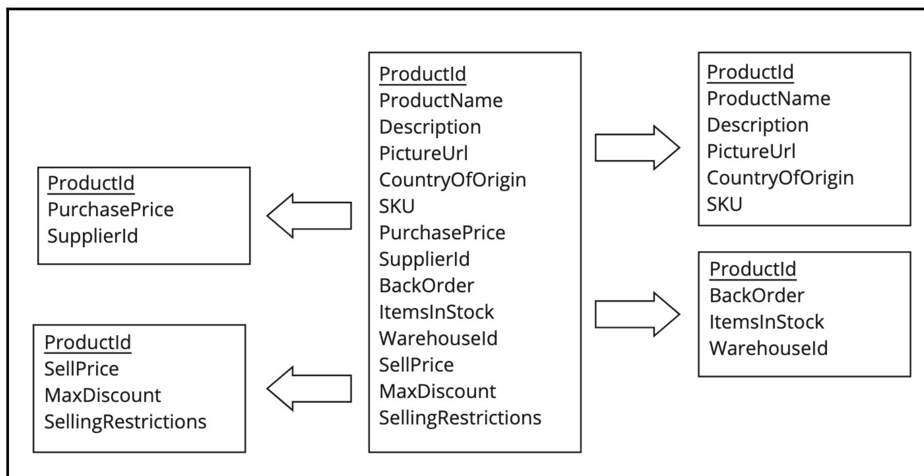
Although we see the events being placed around the timeline, there are deviations in where precisely the sticky notes were placed by different people. It seems that our experts have different areas of interest. Those who are mostly interested in the core business concentrate their attention on features. Developers tend to care about users, logins, and other technical aspects, although it might be irrelevant at the beginning. But those who invest in the new product are very much interested in how and when will they get their money back. I remember that many times when we were doing workshops together, Alberto Brandolini stressed that developers always forget about the money! You can do a small experiment and do an EventStorming session for a new cool feature to see how many developers will think about the commercialization side of it. I bet you'd be surprised.

So, as you can see, the question of *when to split the system* or even when to consider such a split has multiple answers and it is all, unsurprisingly, context-specific. My last, but not least, piece of advice from my own experience is to follow the pain. If you feel awkward and uncomfortable most of the time that you spend writing the code or making designs for a system that you work on, wrong boundaries might be there to blame and to fix. Merge conflicts, excessive coordination, a backlog full of issues that are blocked by dependencies, a class or a data model where everything references everything, situations when touching one thing will most definitely crash something at the other side of the world – all those are clear signs that the system design needs serious attention.

What to split

This section will be short, because we already went through quite a few examples of what belongs together and what doesn't in `Chapter 12`, *Bounded Context*. One of the cases that I can give you here is how can we identify alien concepts in some entities that can provide us with a clue of incorrect system boundaries in the first place. We can use those clues to find better boundaries after doing some knowledge gathering and analysis.

In this example, we get back to one of the most notorious terms that exists: the mighty `Product` entity. Most of us have seen it at some point in our career. The most potent of them all is definitely the `Customer` entity, but let's not go too deep in the rabbit hole. Here is how the `Product` entity can be considered for a split, based on the speed of change of its properties and the language context in different domains:



There's no single product, but multiple views of it

Since the example is quite stereotypical, all of us can spot the obvious. One entity in the system is taking too much responsibility, and, as a result, has quite a lot of dependencies and dependants. That's even without mentioning that the product name never changes, together with the number of items in stock, but, too often, you can see both of those fields on one **Edit product** screen with a single **Save** button.

When you decide where to start in figuring out the new system boundaries in an existing system, I could suggest a few things.

First, talk to people – especially your customers and especially those larger ones where different people execute different functions. Business calls these **departments**. Observe their work and listen to the words that they use to describe things they do in your system. Not only will you be amazed by all those crazy things that users do with your system, but you'll also find out that half of the system features aren't even used, and the other half is somewhat misused. The most important lesson, however, would be the domain knowledge. If people manage to use the system in different departments, they probably also use different parts of it. If one department uses one or two forms and the other department uses another three forms, you might find out that all of those forms are displaying data from the same source, but the shown fieldset is not the same in each case. It would be a clear indication of where to put your attention.

Second, look at your database model and spot places where you can't even see things if you try to re-engineer the model to a diagram in some database management tool because too many arrows are pointing to one or two tables. Too many dependencies is a red flag; that is where you need to put your attention and figure out whether that specific entity takes too much responsibility and might be crying out to be separated, following the proper context boundaries.

The third and the final point for this section is to, again, look for dependencies and coordination between teams. Enough has been written in this book about team autonomy and the disastrous consequences of continuous and tedious coordination. It might not always be possible to influence the structure of the whole organization, but that might not even be required. You can always stress the importance of autonomy to teams so that they can be more productive. One of the things that can help here is the so-called **Inverse Conway's Maneuver**. As you can imagine, it is directly related to Conway's law. If you have a setup where your frontend and backend developers are in different teams (I've seen it way too many times) and continuously blame each other for not delivering in time, you might try applying the maneuver. If you can see at least some context boundaries lurking in the system where people from both teams need to work closely together, bring them together. So, instead of the front team and the backend team, suggest trying with the `Order` handling team and the `Warehouse` management team, for example. The main thing here is that no more than one team is working on one context, although one team can work on more than one context. Things work as long the context boundaries don't leak across team boundaries.

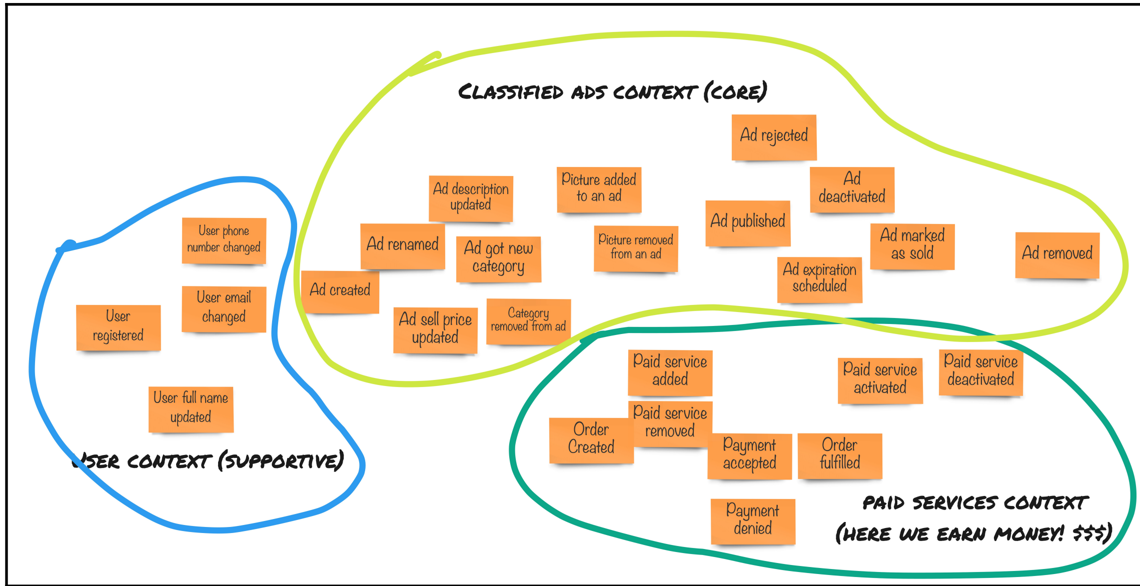
Emerging boundaries continued

Now, if we get back to the emerging boundaries of our sample application, we can observe that there are some spots that speak different languages to us, even on the preliminary model. The main event stream is all about the core features of the product – publishing classified ads to sell the stuff they don't need. When the latter happens, we'll have two happy customers, the buyer and the seller, who met each other using our brilliant system. We can also try to classify this part as being between *complicated* and *complex* using the Cynefin framework.

Another spot on the left of the following image is all about users, logins, and profiles. It is essential, but not as much as the core thing. This aspect of the system can be seen as *simple* if we consider that the best practices exist and there are off-the-shelf products that can help us to get it done. Make no mistake though, the security aspect, in general, is quite complicated; but again, using something pre-made often makes our life much more comfortable.

The third and the final part of the model is all about money. Finally, we get to something that we can monetize. Our experts might not have an idea of what they mean precisely by *paid services*, but we could imagine that to be something like keeping an ad at the top of the search results for a period of time, or making it more visible by applying some effects to how the ad is being rendered on the page.

None of those three areas intersect. It is not always the case, because, again, the context plays its crucial role and we might have the same terminology in different parts of the system, where the meaning of words change. We'll get back to such a scenario in a minute. Now, let's look at how we can initially draw context boundaries based on the observations that were made during the design session:



Contexts emerging from the EventStorming meeting

Getting back to the artifact from your EventStorming session is one of the simplest ways to initially draw the context boundaries. This could happen when the participants of the workshop know the business well enough and when the business itself is not too cumbersome. In many cases, context boundaries aren't easy to find, and you then get back to the speed of change and linguistic boundaries described in the previous chapter.

How to split

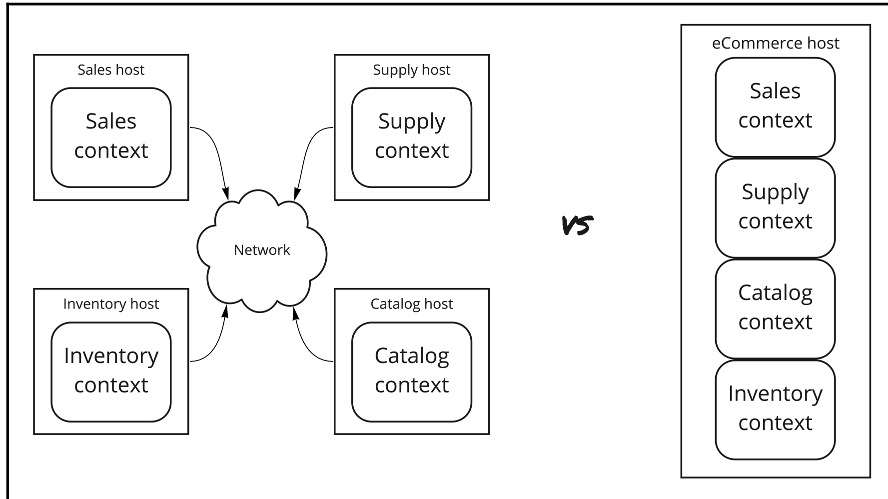
As developers, we usually prefer having technical discussions over talking about the business, at least when we are at work. So, let's now dive a bit deeper into the tech.

As I mentioned before at the beginning of this chapter, the rise of microservices brought up the discussion of how systems should be built on the next level. Many have tried to jump on the hype train and then failed. I often hear that working with a monolithic system makes our lives easier. We can trace dependencies, do step-into debugging, and trace the input-output of the whole application quite easily. All those statements are correct. Does it mean that we should keep building monoliths? The classic answer here is: it depends.

Eric Evans did a talk called *"DDD and Microservices: At last, some boundaries!"* back in 2015. You can easily find this on YouTube, and I strongly recommend that you watch it. Surprisingly, Eric didn't give a piece of strong advice to go on to start implementing microservices if you want to embrace the concept of Bounded Contexts. But the idea remains that physical separation of the code brings an immediate benefit of having the most robust boundaries possible. You can't call something that you are not suppose to call, because that thing is out of your immediate reach. And, if you try to go around this limitation by creating a call over the network, the amount of work associated with this task will probably force you to rethink whether you are doing the right thing at all.

At the same time, as I already cited Simon Brown at the beginning of this chapter, what makes us think that microservices can save a system that is suffering from spaghetti code in any way? Simon's statement makes perfect sense if you think of starting a new product or trying to isolate some things in an existing system on a smaller scale, and the team you have is not very experienced in building distributed systems.

I have quite a lot of experience in building distributed systems, so I know, to some extent, how to avoid or overcome most of the typical issues that developers struggle with when they try to move over to microservices without much prior knowledge and experience. At the same time, without having someone to advise you, I would avoid starting a larger-scale project that involves building many components before getting trained on a small-scale prototype. Instead, you can put your effort into making the system modular while keeping it in a single repository and running it as a monolithic deployable unit:



Modular monoliths allow composition without distribution

In the next section, I will guide you through an example of composing a system with separate modules that can be developed by several developers working independently. I must warn you that, although this approach requires no knowledge of how to build distributed systems, it doesn't guarantee such firm boundaries as multiple isolated components would give you. Developers would still be able to abuse the system structure by directly referencing and calling things across components. Building a modularized system in a single repository requires a high level of discipline from the team. To avoid the system becoming a mess and moving quickly toward the big ball of mud pattern, embrace code reviews with pull requests and try to do pair-programming as much as you can.

Multiple views on the system design

In this section, we'll go through some critical aspects of building a modular system that is still being executed as a single application. Our aim, however, would be to be able to split the system further to multiple smaller applications if we feel a need for it. We will also look into some details about the system and functional design when it comes to context boundaries in practice.

I am unable to show the progress of building a more extensive system for our example domain due to the apparent constraints of the book size and the amount of code. At the same time, the book code repository contains the full implementation of the entire application and all code samples for this section are taken from that repository. I encourage you to spend more time exploring the code and find more implementation details.

Design process

We have been moving along with building an application for our example domain for quite a while by now. We started with sticky notes in `Chapter 4`, *Designing the Model*, implemented the basic domain model in `Chapter 5`, *Implementing the Model*, and then made the system more and more functional. This final chapter will capitalize on most of those deliverables, but we need to get into more details on several levels. The first level is the design process, and we need to go back almost to the same stage as `Chapter 4`, *Designing the Model*, and `Chapter 5`, *Implementing the Model*, to see whether we can find new boundaries considering the overall model that I showed you in the *Emerging boundaries continued* section earlier in this chapter.

While writing the code for our system, we kept using Swagger to call API endpoints to execute operations by sending commands and retrieve data by using queries. This approach is perfectly fine at the start, but no one would seriously consider that our users will use the API. What we need is a proper user-interface for the browser. When it comes to the UI/UX design, backend developers, presumably the primary audience for this book, often feel out of their comfort zones. We believe we can create beautiful code, but when it comes to creating beautiful screens, things get complicated. So, we tend to bring specialists to the product or feature team who can produce quality **user interfaces** (**UIs**). At the same time, those experts are too often working outside of the group or not in a close enough proximity to it. And by *close enough*, I literally mean *inside the team*. It is tough to build a proper system without considering the frontend as an integral part of the system. In fact, UX models have a significant impact on functional requirements for the system behavior as such. Let's have a look at what we can find while working together with UI experts during the design process.

Imagine that the team has discussed how the application frontend would look like, and has come up with the first wireframe:

Classified ad creation – the first screen

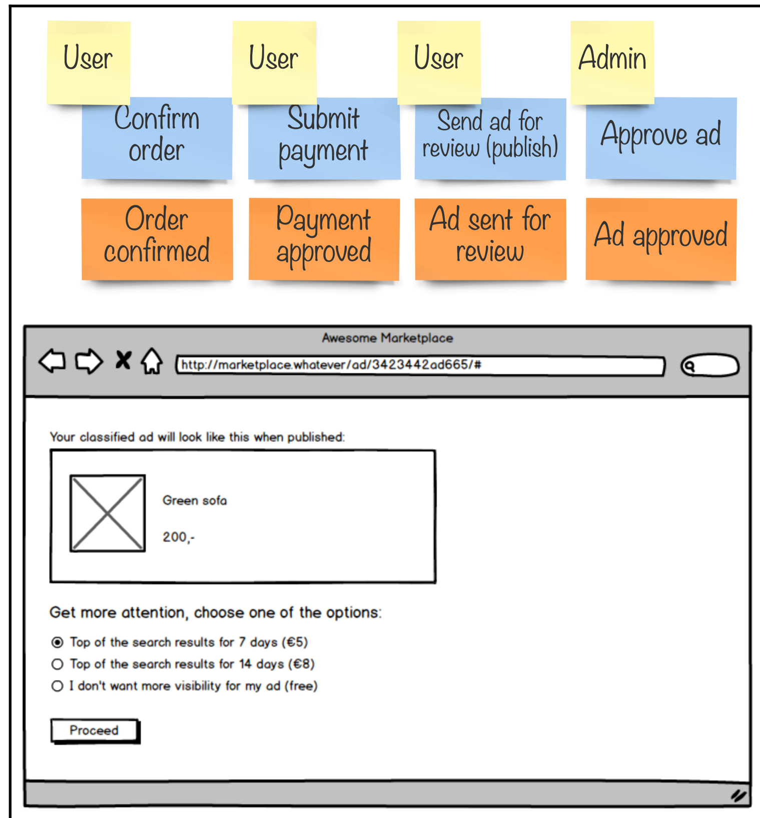
This wireframe already gives us some idea of what actions we need to implement in the domain. It also clearly shows what information is being collected and what the API will get as commands. If we bring the EventStorming artifacts into the same room where the team discusses wireframes, both models start to complement each other:

Wireframe complemented by events it produces

When people with different specializations are working together, they can share knowledge and adjust the way they think toward the same goal, considering different views of the system that they might previously have not even thought of. In the preceding wireframe, we can see that, although the form only has two buttons, **Add an image** and **Next step**, the form itself produces more event types than two. Developers that were working on the domain model need to explain what benefits the system gets when events are more granular, and state transitions are limited to a single operation at a time.

Having such a design for the frontend, when you have a single button but need multiple events, is not a real issue. The code repository for this chapter contains a working example, where commands are being sent to the API when the input fields lose focus, and the value in the field has been changed.

Now, let's see what happens next. If we look at the event flow on our stickies board, the process seems obvious – we need to give the seller an ability to publish the ad and wait for buyers to get in touch. But, aren't we forgetting about the money again? The whole process of adding (optional) paid services before the ad gets published seems to have not been considered, and, without it, our company won't earn any money. So, our designers brought up one more wireframe to address this issue, since we prefer to create a strong link between the user interface and the system backend by making the business flow more explicit using domain events. When team members brainstorm not only the form, look, and feel, but also its behavior, they are able to produce a new artifact:



The monetizing screen with events, commands, and roles applied

The form wireframe is now combined with even more elements, such as commands that produce events, and user roles that will execute those commands. At this stage, we get a better alignment among the team members. Everyone can now understand how UI elements are going to trigger different pieces of the system behavior behind the scenes.

Adam Dymitruk has developed a successful method to design complex systems by using the initial elements of EventStorming combined with real-world UI wireframes, on a one-timed sequence. He calls this technique *event modeling*, and is planning to write a book about it. Although I haven't used this method yet, it looks and sounds very promising, so I strongly recommend looking into it.

You might be wondering by now what all of this has to do with context boundaries? The answer can be found at the event model from the wall. We've seen that the paid services are somewhat separated from the rest of the flow, and that gives us a sign that context boundaries might emerge there. User interface screens are taking the lead in showing us what data needs to be available at different steps of the classified ad publication process. As you will remember from `Chapter 9`, *CQRS – The Read Side*, we create read-models to satisfy the need of the user interface and we strive to design projections to include just as much data as we need to show on a single screen.

By looking at wireframes, we can identify what read-models we need to build and where the data for those system elements might come from. We can also identify what data pieces will be used to make decisions and what elements are purely informational or decorative. At the first wireframe, we've seen everything related to the classified ad entity – the title, the description, the price, and so on. On the second screen, however, we see that the classified ad view almost disappears. It is only presented there to show what was done before, but this information is arguably needed to make a decision to purchase additional visibility for the ad. In fact, many similar services don't even show the publication when offering to add more visibility to it. The information required for the customer to make a decision to pay a small sum to make the ad more visible is solely based on the service description and price. If we are able to immediately show the effect of the selected service on how the ad will look like when it gets published, it is a nice-to-have feature, nothing more. We could conclude from this that the classified ad context is not involved in the process of ordering additional services and making the payment.

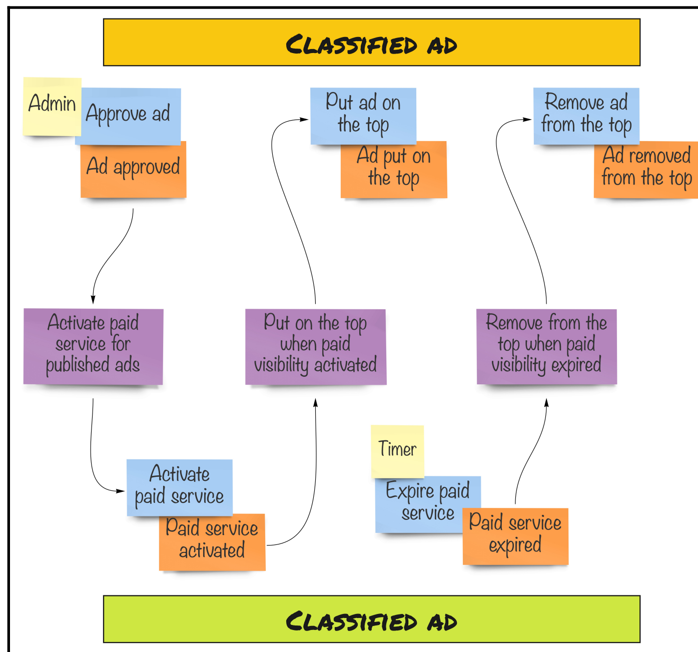
Splitting the entity

We went through the process of writing quite a lot of code for a single context of classified ads from `Chapter 5`, *Implementing the Model*, and beyond. Now we are adding a new concern and we might have a natural urge to complement the existing entity with more information just because it's already there. The code is written, our entity already gets persisted properly, and we have APIs to execute commands and retrieve some data using queries. Adding a couple of new properties is certainly not a big deal, so we can get a quick win in a very short time.

The approach of expanding existing entities is valid at the prototype stage, but remember that it will lead you to the inevitable endless growth of a single entity type, which will quickly get tons of dependencies and become an undeniable mess. Take a look at the example of `Product` that I presented earlier in this chapter. When more people start working on the same code, things will get complicated and start to break. There is no sustainable way to avoid this path unless you are able to identify that different parts of this entity change at different speeds for unrelated needs. When you think about it, you might get reminded about the famous **single responsibility principle**, one of the SOLID principles, which states the following: *A class or module should have one, and only one, reason to be changed*.

When we apply this principle to our entity, we can clearly see that different concerns should not be mixed in one object, although it represents the same concept of the real world. I mentioned this phenomenon in `Chapter 4`, *Designing the Model*. The model does not represent the real world, but only the part of it that is needed to implement a certain business capability or behavior.

Looking from that perspective, we could come up with a model that shows two different sides of the classified ad instance:



One entity becomes two; each part is responsible for its own behavior

The opportunity to make aggregate and context boundaries more explicit is hard to miss. Keeping this in mind, we can start implementing the new functionality in a different component. Because we don't feel a need to distribute our application yet, we can safely start by creating modules in the existing application.

Building the modular system

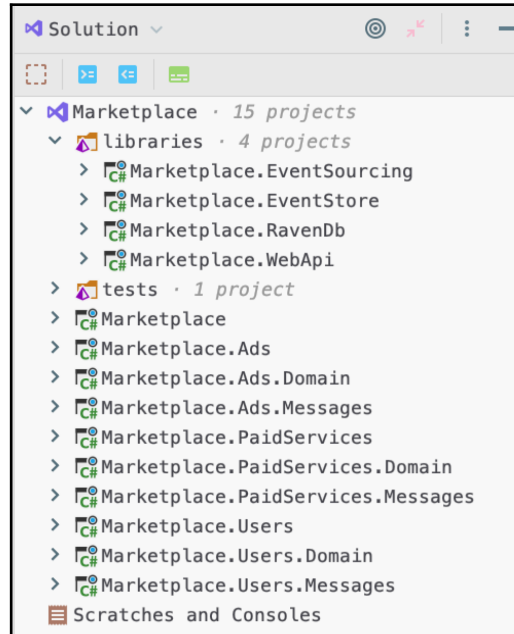
In this section, we will again start writing code to complement the sample application with new functionality. For the new module, I will use a number of techniques to implement the same concepts that you already learned in all previous chapters. The reason that I have changed the implementation style is to show that different ideas can be applied when you have several functional modules, even if they are all physically located in the same repository and hosted within one executable.

The approach I will be using in the new code can be seen as being more functional. There will be fewer classes and many more functions. As an object-oriented language, C# doesn't offer any other way to build functions than placing them inside static classes, which you will see clearly in the code. Functional code is known to be more reliable and easier to run in multiple threads and processor cores due to the absence of the side effects of pure functions that always return some output, and never mutate the input. Again, due to some severe limitations of the language, you will still see some mutations in my functions. The cause of this is mainly that C# still lacks record types and the usage of structs is very limited. Hopefully, we will get record types in C# later in 2019, when new versions of the language will be released.

Modules inside one solution

Since we are separating the system into modules, we could also move the UserProfile module we had before in the main application and create an isolated module for it as well. We could analyze our experience with the system already to see that the user profile information is not being used in the core domain to make any decisions. It is true that we check who can execute certain commands, such as approving the classified ad to be published, but that concern is only related to the authentication context and has nothing to do with the user profile.

Keeping that in mind, we can create several projects to represent modules of the system. When the groundwork is done, the new solution structure looks quite different from what we've seen before:



A solution organized with modules

For convenience purposes, I moved all technical libraries to a solution folder. Projects in this folder don't change that often and the solution folder can, therefore, remain collapsed most of the time.

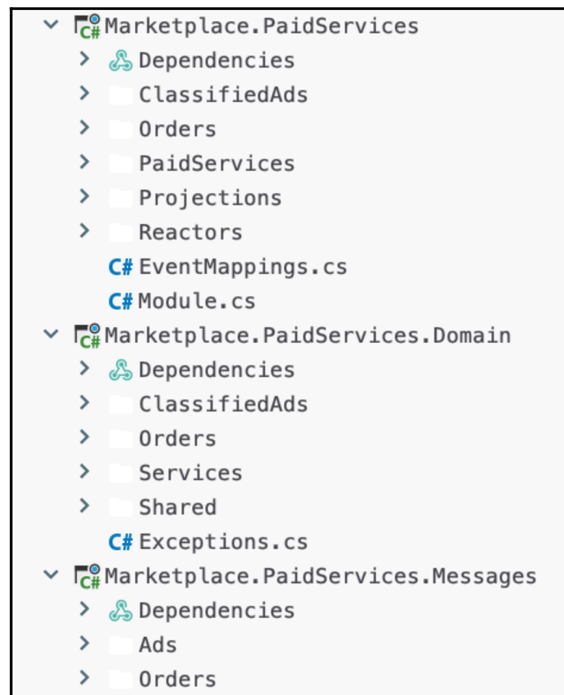
You can see that each proposed component consists of three projects. We can look at one module to understand the structure of it:

Project name	**Purpose**	**References**
`Marketplace.Ads`	Serves as part of the edge, the outer layer of the ports, and adapters architecture	• Infrastructure packages, such as database clients and ASP.NET Core MVC • The domain model project • The project with message contracts
`Marketplace.Ads.Domain`	The domain model itself, with entities and value objects	• The project with message contracts • Necessary abstractions from the `EventSourcing` library

`Marketplace.Ads.Messages`	Keeps all contracts for the domain that are used to communicate to the outside world and within the module itself	• Nothing

So, after slicing our application into modules, we got three isolated components. Each of those components consists of three projects and we still keep our contacts out of any external references and our domain model clean of dependencies to the infrastructure.

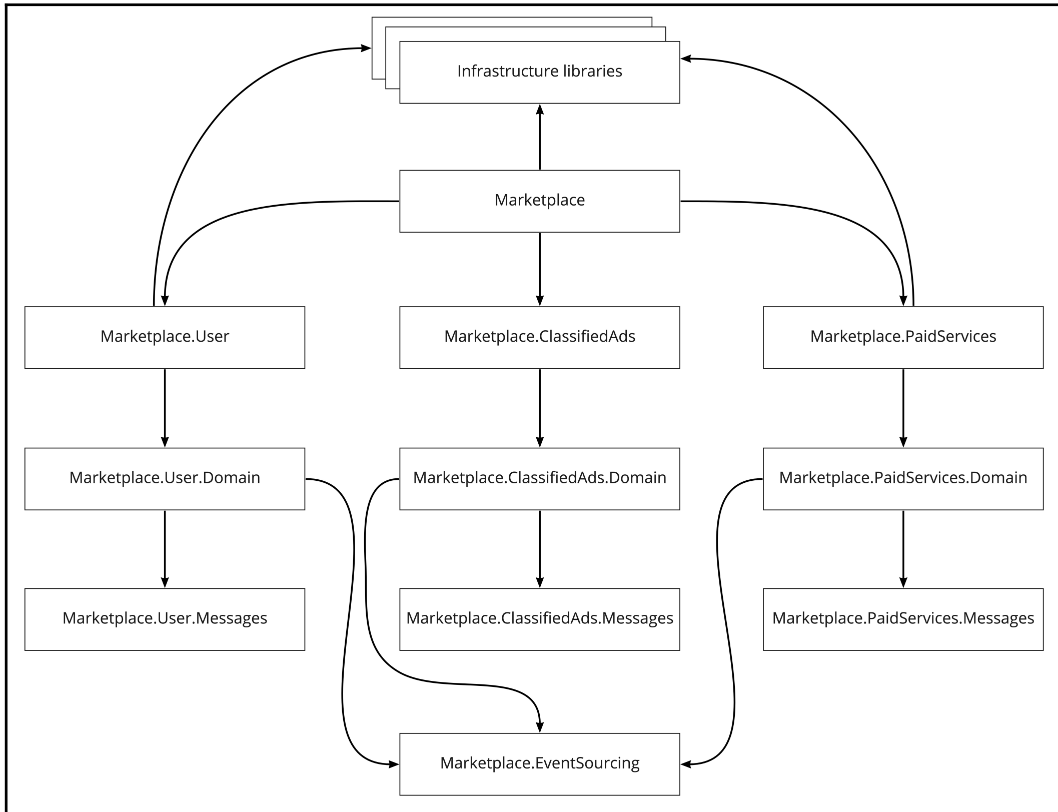
We can now take a look at the structure of one module that consists of those three projects:



Three projects implement everything for a single context

Each project is still organized around its own internal components. Here, you can see that the paid services context has internal modules for the list of services, orders, and its own representation of the classified ad.

The most important thing to mention about the new solution structure is that modules should **never** reference other modules. The overall dependencies can be represented by a simple diagram:



Modules do not reference other modules

In the next section, we will dive into one of the modules to see how it is organized and how it exposes itself to the world.

Module structure

Because our modules are identically organized, it would be just enough to go through the structure of one module to understand the concept. After we analyze the module internals, we'll be able to see its interfaces and how all modules are wired up to the executable application.

We will travel the dependency tree from the bottom level, starting from the project that has zero dependencies and then following arrows on the preceding diagram until we reach the system boundary.

Message contracts project

Let me start with the simplest part of the module – the `Messages` project. This contains all domain events and commands, which, as we discussed in `Chapter 6`, *Acting with Commands*, and `Chapter 10`, *Event Sourcing*, only consist of primitive types. We might notice some complex types there as well for the purpose of having collections. Those complex types never have any logic and the main purpose for all message types is to hold the information that needs to be transferred between components inside the module, to be persisted (events) and used for making requests to the system (commands).

The main reason to keep message contracts in a separate project is that you might need to share those contracts so that other systems can, for example, send you some commands. Remember that the Web API is just one of the possible adapters for your application services, which represent ports. You might also send commands via gRPC or a message bus. The only thing you need to worry about is that all classes that are used as messages can be serialized and deserialized. For that purpose, they all have no advanced constructs such as private constructors and read-only properties. Certainly, there are ways to make messages immutable, as they are supposed to be. The application pipeline should normally not be able, for example, to change the content of a command, so it arrives in the application service with exactly the same data as it got on the application edge. However, making data contract classes immutable might not play well with some serialization libraries, or may require advanced settings to make it all work. You, as a developer, need to choose your own poison and either go for purity or for practicality. Such a choice mainly depends on the skill level and discipline of the development team.

Domain module project

The second project implements the domain model of one Bounded Context. If we look at the `MarketPlace.PaidServices.Domain` project, we will find folders that consolidate several classes related to a single aggregate in this context. For example, the `ClassifiedAds` folder contains the following classes: the `ActivePaidService` value object, the `ClassifiedAd` aggregate, the `ClassifiedAdId` value object, and the `ClassifiedAdState` aggregate state class.

I have already mentioned that I applied some functional techniques in the new project, and now we can have a brief look at the aggregate code.

With the new structure, the aggregate is split into two parts. One part is the aggregate state. It keeps all values that are required for the aggregate instance to take decisions, therefore, it has no power to emit events, but knows how to change them when events are being applied. The state object is also in charge of taking care of its own validity, so when we try to apply an event that makes the state invalid, an exception will be thrown. So, the state is also immutable. Instead of mutating its own properties, the `When` method returns the new instance of the state with new values. It gives us the ability to empower the state validation to compare the previous state with the new state. It is very useful when the new state seems to be valid, but the state transition itself is not correct, and, to validate this, we need to validate the state transition as a whole:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Marketplace.EventSourcing;
using Marketplace.PaidServices.Domain.Services;
using static Marketplace.PaidServices.Messages.Orders.Events;

namespace Marketplace.PaidServices.Domain.Orders
{
    public class OrderState : AggregateState<OrderState>
    {
        public override OrderState When(
            OrderState state,
            object @event
        )
            => With(
                @event switch
                {
                    V1.OrderCreated e =>
                        With(state, x => x.AdId = e.ClassifiedAdId),
                    V1.ServiceAddedToOrder e =>
                        With(
                            state, x => x.Services.Add(
                                PaidService.Find(e.ServiceType)
                            )
                        ),
                    V1.ServiceRemovedFromOrder e =>
                        With(
                            state, x => x.Services.RemoveAll(
                                s => s.Type ==
                                     PaidService.ParseString(e.ServiceType)
                            )
                        ),
                    V1.OrderFulfilled _ =>
                        With(state, x => x.Status = OrderStatus.Completed),
```

```
                    _ => this
                },
                x => x.Version++
            );

        protected override bool EnsureValidState(OrderState newState)
            => newState switch
                {
                    { } o when o.AdId == Guid.Empty => false,
                    { } o when o.Status == OrderStatus.Completed
                               && o.Services.Count != Services.Count
                               => false,
                    _ => true
                };

        internal double GetTotal() => Services.Sum(x => x.Price);

        internal Guid AdId { get; set; }
        OrderStatus Status { get; set; } = OrderStatus.New;
        internal List<PaidService> Services { get; } = new
List<PaidService>();

        internal string[] GetServices()
            => Services.Select(x => x.Type.ToString()).ToArray();

        enum OrderStatus { New, Completed }
    }
}
```

In the preceding code snippet, I applied such a check to ensure that the collection of services remains unchanged for a completed order. This check follows the business requirement that when the order is finalized and paid, users cannot add or remove services for that order.

To execute actions for the `Order` aggregate, I created a module with functions that emit new events and apply them to the `OrderState` object. As I mentioned before, the only way to implement modules with functions in C# is to create a static class. Here, you can see the code for the `Order` module:

```
using System.Linq;
using Marketplace.PaidServices.Domain.ClassifiedAds;
using Marketplace.PaidServices.Domain.Services;
using Marketplace.PaidServices.Domain.Shared;
using static Marketplace.PaidServices.Messages.Orders.Events;

namespace Marketplace.PaidServices.Domain.Orders
{
```

```
public static class Order
{
    public static OrderState.Result Create(
        OrderId orderId,
        ClassifiedAdId classifiedAdId,
        UserId customerId
    )
        => new OrderState().Apply(
            new V1.OrderCreated
            {
                OrderId = orderId,
                ClassifiedAdId = classifiedAdId,
                CustomerId = customerId
            }
        );

    public static OrderState.Result AddService(
        OrderState state,
        PaidService service
    )
        => state.Services.Contains(service)
            ? state.NoEvents()
            : state.Apply(
                new V1.ServiceAddedToOrder
                {
                    OrderId = state.Id,
                    ServiceType = service.Type.ToString(),
                    Description = service.Description,
                    Price = service.Price
                },
                new V1.OrderTotalUpdated
                {
                    OrderId = state.Id,
                    Total = state.GetTotal() + service.Price
                }
            );

    public static OrderState.Result RemoveService(
        OrderState state,
        PaidService service
    )
        => !state.Services.Contains(service)
            ? state.NoEvents()
            : state.Apply(
                new V1.ServiceRemovedFromOrder
                {
                    OrderId = state.Id,
                    ServiceType = service.Type.ToString()
```

```
                    },
                    new V1.OrderTotalUpdated
                    {
                        OrderId = state.Id,
                        Total = state.GetTotal() - service.Price
                    }
                );

        public static OrderState.Result Fulfill(OrderState state)
            => state.Services.Any()
                ? state.Apply(
                    new V1.OrderFulfilled
                    {
                        OrderId = state.Id,
                        ClassifiedAdId = state.AdId,
                        Services = state.Services
                            .Select(
                                x => new V1.OrderFulfilled.Service
                                {
                                    Description = x.Description,
                                    Price = x.Price,
                                    Type = x.Type.ToString()
                                }
                            )
                            .ToArray()
                    }
                )
                : state.Apply(
                    new V1.OrderDeleted
                    {
                        OrderId = state.Id
                    }
                );
    }
}
```

Functions in the module are taking decisions based on the information available in the state object. For example, when the user tries to add a service that has been already added to the order, the operation would not be completed. I don't really want to throw an exception here, because such an action is not really expected to happen. Our business requirement states that orders may only contain one service instance of a single kind, so users cannot add the *show on top of the search results* option twice, because it doesn't make sense. The UI should prevent this from happening, but, in case such a command still goes through and comes to the domain, it will just be ignored. The same approach applies to the removal of a service. If we eventually receive a command to remove a service that cannot be found in the order, we will just ignore it.

You have noticed that now we have two different aggregates for `ClassifiedAd` in two domain projects. If you remember the picture from the *Splitting the entity* section, we have modeled the system in a way that two Bounded Contexts need to have their own view on the same object and each of those views will be taking care of its own concerns. So, for the paid services context, we need only to keep the information of all active services that were ordered by the customer, but we have no interest in other information about it. The only thing that those two representations of the same real-world entity share with each other are the identity of that entity. If we have one real-world entity in the system, it must be uniquely identified across the whole system by sharing its identity. We do not enforce any kind of referential integrity here since we don't have one single data model to persist with our entities. Therefore, all components of our system need to ensure their own consistency inside their boundaries.

By now, you might be wondering how those isolated components communicate if they cannot call each other directly. All transactions only happen for a single aggregate. Therefore, all cross-contexts, and sometimes cross-aggregate communications, happen using domain events. Such communication is considered as **integration** and I will cover this important topic in the final section.

Application project

Finally, we are getting to the application project that hosts application services and interfaces that are exposing our domain to the outside world.

Let's have a look at the application project for the paid services context. It is also split into modules, with each of those modules taking care of exposing one aggregate from the domain project using application services and APIs. Application services and command APIs are very similar to those that we have already seen in the previous chapter. Following my functional approach for aggregates, I had to make some changes in the application service code as well. For example, the command service for orders now looks a bit different if we compare it with the command service for classified ads from the previous chapter:

```
using System.Threading.Tasks;
using Marketplace.EventSourcing;
using Marketplace.PaidServices.Domain.ClassifiedAds;
using Marketplace.PaidServices.Domain.Orders;
using Marketplace.PaidServices.Domain.Services;
using Marketplace.PaidServices.Domain.Shared;
using static Marketplace.PaidServices.Messages.Orders.Commands;

namespace Marketplace.PaidServices.Orders
{
    public class OrdersCommandService : CommandService<OrderState>
```

```
    {
        public OrdersCommandService(IFunctionalAggregateStore store)
            : base(store) { }

        public Task Handle(V1.CreateOrder command)
            => Handle(
                command.OrderId,
                state => Order.Create(
                    OrderId.FromGuid(command.OrderId),
                    ClassifiedAdId.FromGuid(command.ClassifiedAdId),
                    UserId.FromGuid(command.CustomerId)
                )
            );

        public Task Handle(V1.AddService command)
            => Handle(
                command.OrderId,
                state => Order.AddService(
                    state, PaidService.Find(command.ServiceType)
                )
            );

        public Task Handle(V1.RemoveService command)
            => Handle(
                command.OrderId,
                state => Order.RemoveService(
                    state, PaidService.Find(command.ServiceType)
                )
            );

        public Task Handle(V1.FulfillOrder command)
            => Handle(command.OrderId, Order.Fulfill);
    }
}
```

Each method still handles one single command and calls a function from the `Order` module. The application service base class takes care of retrieving the aggregate state from the store and then saving new events to the aggregate stream. It might happen that the function returns zero events, for example when we try to add a service to the order twice. You can refer back to the `Order` module code to see that, in that case, the function will return a special result, `state.NoEvents()`. The application service will ignore such a result and the aggregate stream will not get any new events.

Going functional

Since we started to analyze how the functional approach is different from the object-oriented one, we will now have a brief look at the `ApplicationService` base class to see how it works:

```
using System;
using System.Threading.Tasks;

namespace Marketplace.EventSourcing
{
    public abstract class CommandService<T>
        where T : class, IAggregateState<T>, new()
    {
        IFunctionalAggregateStore Store { get; }

        protected CommandService(IFunctionalAggregateStore store)
            => Store = store;

        protected async Task Handle(
            Guid id,
            Func<T, AggregateState<T>.Result> update)
        {
            var state = await Store.Load<T>(id);
            await Store.Save(state.Version, update(state));
        }
    }
}
```

As you can see, the base class code is extremely simple. It has the advantage of the aggregate state being immutable, and, instead of retrieving new events from the aggregate itself using the `GetChanges()` method, as we did in the previous chapter, it uses the function result that already has a collection with new events.

This approach clearly shows the essence of the command execution process in Event Sourcing, the command execution processes described as $e_{n+1} = f(e_1 \ldots e_n, c)$. When we apply a command to the collection of events that were previously stored as an aggregate, we get a new event back. So, we can get more than one event as the result of a single operation and we must ensure that our Event Store is capable of storing all new events in a single transaction to ensure consistency.

To complete the picture, we might look at the code for the new aggregate store class:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;
using EventStore.ClientAPI;
using Marketplace.EventSourcing;

namespace Marketplace.EventStore
{
    public class FunctionalStore : IFunctionalAggregateStore
    {
        readonly IEventStoreConnection _connection;

        public FunctionalStore(IEventStoreConnection connection)
            => _connection = connection;

        public Task Save<T>(
            long version,
            AggregateState<T>.Result update
        )
            where T : class, IAggregateState<T>, new()
            => _connection.AppendEvents(
                update.State.StreamName, version, update.Events.ToArray()
            );

        public Task<T> Load<T>(Guid id) where T : IAggregateState<T>, new()
            => Load<T>(id, (x, e) => x.When(x, e));

        async Task<T> Load<T>(Guid id, Func<T, object, T> when)
            where T : IAggregateState<T>, new()
        {
            const int maxSliceSize = 4096;

            var state = new T();
            var streamName = state.GetStreamName(id);

            var position = 0L;
            bool endOfStream;
            var events = new List<object>();

            do
            {
                var slice = await _connection
                    .ReadStreamEventsForwardAsync(
                        streamName, position, maxSliceSize, false
                    );
                position = slice.NextEventNumber;
                endOfStream = slice.IsEndOfStream;

                events.AddRange(
                    slice.Events.Select(x => x.Deserialze())
```

```
                );
            } while (!endOfStream);

            return events.Aggregate(state, when);
        }
    }
}
```

The new code is still quite simple and it is also more production-ready. In the previous version, we were reading only 1,024 events from the stream. It was fine for the demo purpose and to keep the code simple, but you can rely on that in production. You have no guarantee that your aggregate stream will always have less than a thousand events. Event Store supports reading events from a stream in slices with the maximum slice size of 4,096 events. So, the new code tries to read the maximum possible slice and does it until the stream ends. The new version is, therefore, able to handle streams of any length.

For production systems that have aggregate streams that can grow over a few thousand events, you might need to apply the snapshot pattern. With snapshots, your aggregate store will check the number of events in the stream and create a special event in different snapshot steam. Such an event will contain the memento object of the whole aggregate state. When you then need to read the state, your aggregate store will first try to get the last snapshot and then read all events that were applied after the snapshot was created.

The aggregate version, which represents the event position in the stream, is the essential property that allows you to find the starting position for reading events after the last snapshot, so you must include the aggregate version in the snapshot event as well.

The final piece of the functional aggregate implementation is the base class for the aggregate state. The code for this class can be found in the chapter repository. I don't want to include it here due to its size. The code is purely technical and the main reason for it to be a bit cumbersome is the absence of a proper immutable complex type in the current version of C#. Should the situation improve, I will update the code to embrace new language elements.

However, I would like to show you two pieces of the aggregate state base class:

```
public Result Apply(params object[] events)
{
    var newState = this as T;
    newState = events.Aggregate(newState, Apply);
    return new Result(newState, events);
```

```
}

T Apply(T state, object @event)
{
    var newState = state.DeepClone();
    newState = When(newState, @event);

    if (!EnsureValidState(newState))
        throw new InvalidEntityState(
            this, "Post-checks failed"
        );

    return newState;
}
```

In the preceding code snippet, you can see my attempt to implement state immutability. I had to use an external library called `DeepClone` to create a new instance of the state. With record types, I would just use the assignment operator and most of this code would probably even disappear. Still, this code also implements the other core concept of Event Sourcing, which can be expressed by this expression: $s' = f(s, e)$. The new state is just a result of a function that gets the previous state and applies an event to it.

Bootstrapping modules

The whole Bounded Context module needs to be wired up to the main application executable. We can certainly add all the module wiring code to the application bootstrap, but if we do that, our application `Startup` class will become quite large and cumbersome. We will also have a significant risk of wiring conflicts, were we to use the services collection of ASP.NET Core when we get the wrong dependencies applied to services in the module.

To make the module bootstrapping more explicit, I put the wiring for each module to the application project of the module itself. Here, you can see how the `PaidServices` module is initialized:

```
using System;
using EventStore.ClientAPI;
using Marketplace.EventSourcing;
using Marketplace.EventStore;
using Marketplace.PaidServices.ClassifiedAds;
using Marketplace.PaidServices.Orders;
using Marketplace.PaidServices.Projections;
using Marketplace.PaidServices.Reactors;
using Marketplace.RavenDb;
using Microsoft.Extensions.DependencyInjection;
using Raven.Client.Documents;
```

```
using Raven.Client.Documents.Session;
using static Marketplace.PaidServices.Projections.ReadModels;

namespace Marketplace.PaidServices
{
    public static class PaidServicesModule
    {
        public static IMvcCoreBuilder AddPaidServicesModule(
            this IMvcCoreBuilder builder,
            string databaseName
        )
        {
            EventMappings.MapEventTypes();

            builder.Services.AddSingleton(
                c => new OrdersCommandService(c.GetStore())
            );

            builder.Services.AddSingleton(
                c => new ClassifiedAdCommandService(c.GetStore())
            );

            builder.Services.AddSingleton(
                c =>
                {
                    var store = c.GetRequiredService<IDocumentStore>();
                    store.CheckAndCreateDatabase(databaseName);
                    const string subscriptionName = "servicesReadModels";

                    IAsyncDocumentSession GetSession()
                        => c.GetRequiredService<IDocumentStore>()
                            .OpenAsyncSession(databaseName);

                    return new SubscriptionManager(
                        c.GetRequiredService<IEventStoreConnection>(),
                        new RavenDbCheckpointStore(
                            GetSession, subscriptionName
                        ),
                        subscriptionName,
                        new RavenDbProjection<OrderDraft>(
                            GetSession,
                            DraftOrderProjection.GetHandler
                        ),
                        new RavenDbProjection<CompletedOrder>(
                            GetSession,
                            CompletedOrderProjection.GetHandler
                        )
                    );
```

```
                }
            );

            builder.Services.AddSingleton(
                c =>
                {
                    var connection =
                        c.GetRequiredService<IEventStoreConnection>();
                    const string subscriptionName = "servicesReactors";

                    return new SubscriptionManager(
                        connection,
                        new EsCheckpointStore(
                            connection, subscriptionName),
                        subscriptionName,
                        new OrderReactor(
c.GetRequiredService<ClassifiedAdCommandService>()
                        )
                    );
                }
            );

builder.AddApplicationPart(typeof(PaidServicesModule).Assembly);

            return builder;
        }

        static IFunctionalAggregateStore GetStore(
            this IServiceProvider provider
        )
            => new FunctionalStore(
                provider.GetRequiredService<IEventStoreConnection>()
            );
    }
}
```

The `Module` class has one extension method for the `IMvcCoreBuilder` interface that registers all dependencies in the application container. The main parts in the code of the module are as follows:

- Mapping event types to strings
- Registering application services as singletons
- Registering EventStore subscriptions for read-models and reactors (we'll get back to this in the *Integrations* section)
- Registering all API controllers by calling the `AddApplicationPart` method

The only reason for this method to extend the `IMvcCoreBuilder` interface, instead of `IServiceCollection`, is that the `AddApplicationPart` method belongs to the MVC initialization and we must call it to register our controllers. If you are using another library to build the API, such as Carter or Nancy, you could avoid most of the registration code by providing dependencies for your API modules explicitly.

In this section, we went through all the parts that compose one single module. Each module represents a separated piece of the whole system and implements one Bounded Context. As I mentioned before, the one-to-one alignment of modules with Bounded Contexts is not a requirement, but it is a good start. As soon as context boundaries become clearer, you need to ensure that those boundaries are properly protected. In the next section, we'll bring all the modules together and get our application running.

Building the system

When we have all our modules in place and complete all the code that ensures that the module can be wired up to the application by calling its registration methods, we can bring all modules together and create an application that represents the composition of modules as a system.

Wiring modules to the application

We need to change the `Startup` class and take advantage of separated registration of modules to make it readable and more explicit. The whole class code is still quite long, so I will be including snippets from it and will explain each of them.

Let's start with the `Startup.ConfigureServices` method, which contains most of the bootstrapping code.

Once again, it starts with registering the infrastructure:

```
var esConnection = EventStoreConnection.Create(
    Configuration["eventStore:connectionString"],
    ConnectionSettings.Create().KeepReconnecting(),
    Environment.ApplicationName
);
var purgomalumClient = new PurgomalumClient();

var documentStore = ConfigureRavenDb(
    Configuration["ravenDb:server"]
);
```

```
services.AddSingleton(new ImageQueryService(ImageStorage.GetFile));
services.AddSingleton(esConnection);
services.AddSingleton(documentStore);
```

The preceding code is the same as we had before. The whole application infrastructure is still being initialized at the application level, so modules don't really control what infrastructure components being used. The application controls the configuration and is therefore able to decide how to connect to database servers, which service to use for profanity checks, and so on. We also add the services we require to the application container so they can be resolved by modules if needed.

Next, we register the hosted service that controls the application connection to `EventStore`:

```
services.AddSingleton<IHostedService, EventStoreService>();
```

We were configuring the service explicitly before, but now our modules register subscriptions independently in their own bootstrapping code, so all those subscriptions will be resolved automatically when the application will try to instantiate the `EventStoreService`. The service itself will get a collection of all subscriptions and execute them accordingly.

The final important bit is the MVC wiring:

```
services
    .AddMvcCore(
        options => options.Conventions.Add(new CommandConvention())
    )
    .AddApplicationPart(GetType().Assembly)
    .AddAdsModule(
        "ClassifiedAds",
        new FixedCurrencyLookup(),
        ImageStorage.UploadFile
    )
    .AddUsersModule(
        "Users",
        purgomalumClient.CheckForProfanity
    )
    .AddPaidServicesModule("PaidServices")
    .AddJsonFormatters()
    .AddApiExplorer()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Here, we finally got to the point when we register all our modules. Since we have the wiring code for each module encapsulated inside the module itself, we can simply use those extensions methods that we've created in each module.

Some services, such as the profanity check service of the currency lookup, aren't registered in the container. Not all modules need them and therefore there's no need to make the wiring more complicated than it already is.

You can also find some differences between the new MVC initialization code and the code we used before. I use the `AddMvcCore` method instead of `AddMvc` and have added some other calls, such as `AddJsonFormatter`. The reason for this is that I don't use the full MVC, but MVC Core, which is enough to serve the needs of API controllers. We don't have any views and don't use Razor in the application, so it makes no sense to bring this extensive functionality to the system. As a result of this decision, all API controllers are now inheriting from the `ControllerBase` class. One issue that I've got with MVC Core is that the implicit model binding to complex types, which we used for commands, stopped working. The solution to this issue is to either use the `[FromBody]` attribute for all command API parameters or to use the model binding convention. In the sample code, I used the convention to avoid polluting the API code with attributes. Alternative API frameworks, such as ServiceStack, Carter, or Nancy, always bind complex types correctly and this issue doesn't apply if you use one of those.

Integration

One of the most important aspects of the strategic DDD, when our system becomes a composition of components, is the cross-component communication. Regardless of the composition method, whether it is a single modularized application or a bunch of independent services, we often need those components to talk together. We strictly avoid hard dependencies between Bounded Contexts, but when our components form a system, we need to make the system to work as a whole.

This book covers the strategic design topic very briefly, and only from a practical perspective.

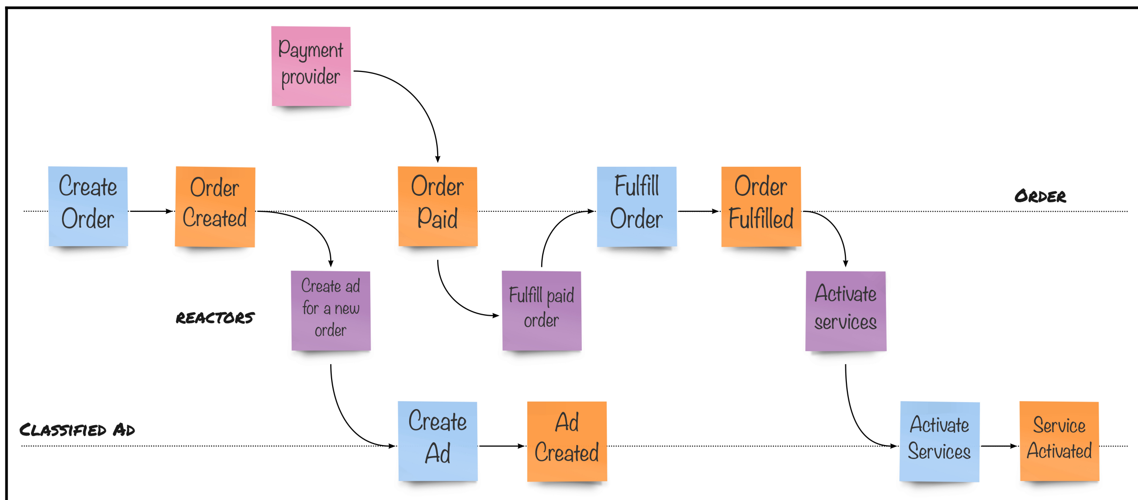
The book *Domain-Driven Design, Tackling Complexity in the Heart of Software*, by Eric Evans, describes all aspects of strategic DDD in details. If you are being serious about taking DDD for a ride, make sure you read *Chapter 14* of Eric's book to understand what types of relations between Bounded Contexts exist, which one you need to choose, and how to implement them.

This book contains many diagrams that show how a single entity like `Product` is split into several aggregates that live in different Bounded Contexts, but we won't seriously expect users of our system to visit the UI of each component and create a separate instance of `Product` in each of them. We agreed that, despite the fact that each context models its own view of an entity, in the real world, only one single instance of that entity exists. For that `Product` example, we need to ensure that when the user registers a certain product in the system for the first time, all Bounded Contexts get informed about this fact and act accordingly.

Because our transactions are always limited to the scope of a single aggregate instance, we cannot execute large-scale transactions across all components of the system to perform such complex operations. Therefore, most of the time the information is distributed within service boundaries by using asynchronous messaging. We already have a few `Messages` projects in our sample application, and integration is the main reason why I added all contact classes to separated projects.

Cross-aggregate integration

Let's have a look at how integration would work in our system. First, we need to look at the integration flow **inside** one Bounded Context. There, some domain events produced by one aggregate that trigger reactions in another aggregate within the context boundaries:



Cross-aggregate integration within one context

The preceding diagram shows parts of the lifecycle for two aggregates – the order and the classified ad. Here, the `ClassifiedAd` aggregate is the paid services context own view on the real-world classified ad. When the seller puts an ad together, the system takes them to the page where they can choose optional services to increase the ad visibility. We have seen some wireframes in the *Design process* section in this chapter. We can also look at the real screen that our system shows to the user for them to create an ad:

The ad creation form

Here, when the seller clicks the **Next** button, we move on to the services screen. At that exact moment, the UI sends a command to the API to create a new order. The command code includes the new order ID (the UI should generate a new GUID), classified ad ID, and the user ID of the seller:

```
public class CreateOrder
{
    public Guid OrderId { get; set; }
    public Guid ClassifiedAdId { get; set; }
    public Guid CustomerId { get; set; }
}
```

A new `Order` aggregate will be created within the `PaidServices` context and we get an `OrderCreated` event in the Event Store. From there, we can start our integration process.

As you have seen on the integration flow diagram in this section, there are a few stickies called *reactors*. Reactors represent what, in the EventStorming world, is called **policies**. They execute domain policies, and, in our case, we have three of them in the paid services domain:

- When a new order is created, we also create an associated classified ad to record all active services for the ad.
- When the order is paid, it needs to be fulfilled.
- When the order is fulfilled, all paid services for the ad need to be activated.

The payment process is handled by an external payment provider, so we need to integrate with an external system. We won't cover integration with external systems in this book, but it is not hard to deduce how it is done from the integration code that is available in the book code. The only difference is that you may or may not get the event in the Event Store. Usually, external systems either call our API and we can emit the event from there, or we would have an API call directly from the UI and then the UI can give us a sign when the payment is processed. In the latter case, it is rarely a good idea to trust the information received from the outside world, unless it comes from a reliable source, and the UI is not one of them. So, after receiving important information from the UI, you'd better go to the payment provider and double-check whether the payment has actually been processed.

For two other cases, we have the following reactor code that ensures that both domain policies get executed:

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Marketplace.EventStore;
using Marketplace.PaidServices.ClassifiedAds;
```

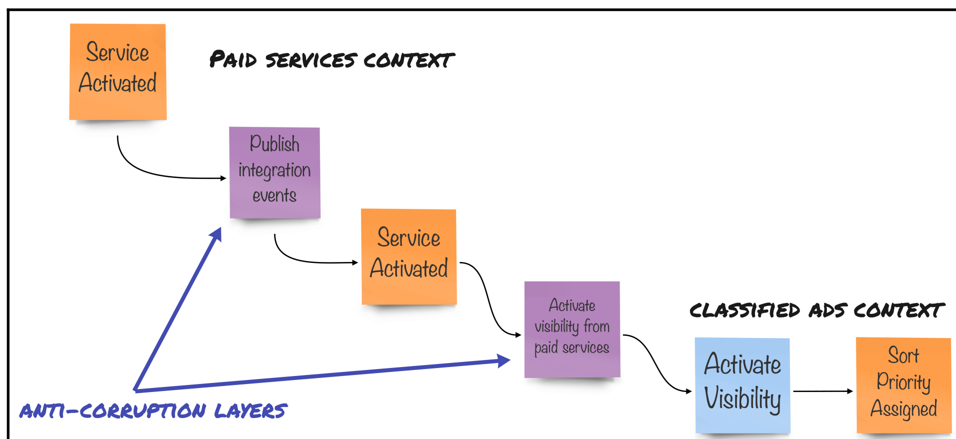
```
using Events = Marketplace.PaidServices.Messages.Orders.Events;
using V1 = Marketplace.PaidServices.Messages.Ads.Commands.V1;

namespace Marketplace.PaidServices.Reactors
{
    public class OrderReactor : ReactorBase
    {
        public OrderReactor(
            ClassifiedAdCommandService service
        ) : base(@event => React(service, @event)) { }

        static Func<Task> React(
            ClassifiedAdCommandService service,
            object @event
        )
            => @event switch
            {
                Events.V1.OrderCreated e =>
                    () => service.Handle(
                        new V1.Create
                        {
                            ClassifiedAdId = e.ClassifiedAdId,
                            SellerId = e.CustomerId
                        }
                    ),
                Events.V1.OrderFulfilled e =>
                    () => service.Handle(
                        new V1.FulfillOrder
                        {
                            ClassifiedAdId = e.ClassifiedAdId,
                            When = e.FulfilledAt,
                            ServiceTypes = e.Services
                                .Select(x => x.Type)
                                .ToArray()
                        }
                    ),
                _ => (Func<Task>) null
            };
    }
}
```

Overall, the integration code is not that different from the projection code. We still get some events and need to react to what happened in the domain. In this case, instead of updating read-models, we tell our application service to execute some commands. By converting events, which represent something that already happened, to commands, we make the intent or our reactions explicit, so our domain model follows the usual command execution process. Because we use the application service, you can imagine that the integration code can be written by reacting to messages that arrive using another transport, such as a message bus. However, for the integration within one context, subscribing to the Event Store is more reliable, because you only get the event after the transaction for the previous command is completed and you only get it once. Message brokers rarely give you guarantees on ordered, one-off deliveries.

It is also perfectly fine to use domain events as message contracts when it comes to the integration inside the context. You, as the context owner, control all of its internals, and therefore changes in message contacts; in this case, events can be done freely without breaking the integration. Remember, however, that changes to events need to be done with care, and here I would again refer to Greg Young's book, *Versioning in an Event Sourced System*.

Cross-context integration

When it comes to the integration between Bounded Contexts, we need to follow the same pattern, but we have a little more to consider. Let's first look at the integration flow in our sample application to understand what and why we need to integrate:



Integration flow between contexts with anti-corruption layers

The diagram shows that, for cross-context integration, things might get more complicated. Let's go through the flow in the following steps:

- The order gets paid and then fulfilled, which triggers the reaction to activate services and the paid services context published to one or more `ServicesActivated` events.
- In the paid services context, we have a reactor that publishes an integration event with the same (or almost the same) content.
- In the classified ads context, we have a reactor that receives this event and executes the proper command. It knows how to convert the activated service type to proper domain internals, such as to show the ad on the top of search results for a week, or to use a large card to show the ad.

Overall, the process is not hard to grasp, but I need to explain a couple of things here.

First, why do we might need integration events? The reason for this is that domain events are internal contracts for the domain. We persist domain events to the Event Store so we can retrieve them later and rebuild the aggregate state. Those events are normally not being exposed to the outside world. The use of domain events is limited by context boundaries. If you look at our new solution structure, you'll find out that projections are also located inside each context and only project events that belong to that context. At the moment we start to share information from the context to the outside world, we lose control on who might consume them. This, by default, makes us responsible for keeping contracts that we expose stable. Commands, for example, are contracts that must be stable and properly versioned for the sake of compatibility for external consumers. When we add a property to some command, we might not need to create a new version of it, but then we must ensure that the information provided in the new property is not required to process the command. If we then, however, start to require more information for a certain command, we need to create a new command instead, which can be either a completely new command or a version of the existing command. When we get a new command, it is important to ensure that both commands are still available for the API consumers, because if we don't know who sends us those commands, we can't just break things by not accepting old commands. It is possible to shut down the old API, but then we must give some time for the API consumers to make necessary changes in their system accordingly. Until that moment, our system needs to support both versions.

The same is valid for events. When we send events for other systems to consume, we need to ensure that we treat it as a contract, so all those concerns that I just described then become valid. You might be okay or not okay with this fact. If you want to expose your system behavior to the outside world so they can react to changes in your context, but you also want to keep the freedom to change your domain events, you might need to consider using integration events. It is not always the case that when you change your domain event the integration event needs to be changed as well. So, when you make such a change and you have integration events, only changes in the publishing reactor would be necessary. Because you control the publishing reactor, all those changes stay within your context boundaries. By applying this method, you prevent your domain from corrupting other systems by changing the public contract at will.

Second, what is the **Anti-Corruption Layer** (**ACL**)? ACL is one of the concepts in strategic design from Eric Evan's DDD book. The reason that ACLs exist is to convert something that happens outside of the context boundary to internal artifacts and vice versa. The integration event publisher can be seen as an ACL that prevents our system from corrupting other systems. When the direction changes, we must protect our domain from being corrupted by other systems. For example, on the preceding integration flow diagram, you can see that the classified ads context has a rector that consumes integration events that come from the paid services contexts and convert them to internal commands and then executes them.

Anti-corruption layers take different forms. For example, you might need to integrate with an external system that doesn't produce any events, but can be called via an API to get some information on a regular basis. In this case, you can build an ACL service that will poll the external system API from time to time and then emit integration events. Another popular pattern is integration with a legacy system. If you have some older system that only has a SQL database and no other public interfaces, you can use table polling or database triggers to discover changes of the information that you need to expose to other systems as integration events, and publish those events as soon as you get new, updated, or deleted records. I build a small library for the MassTransit open-source messaging library, which does exactly that; you might want to look at it if you have a need to integrate with a legacy system. I used this method many times in production systems and it works very well.

Integration events don't need to be put into the event stream. Usually, you would never do such a thing as replaying integration events. I remember when it happened in a system that I participated in building, and that didn't end well. Integration events are usually short-lived and transient by nature, so you can use other technologies such as message brokers to distribute them.

To finalize the integration part, you might be wondering why the cross-context integration code in the sample application is not following the pattern I described. Here, we come to another type of relationship between two Bounded Contexts. Eric Evans described quite a few relationship types to identify and choose from, but I want to mention only one in addition to the ACL pattern. In our case, the application is very small. It is hosted in a single solution and it is likely that not many developers are working with it. In such a case, we can apply the *conformist* relationship type. When one Bounded Context conforms and adopts all the changes that the other context makes in its contracts, the integration is greatly simplified on the code level. Of course, there is a trade-off. You remove one paid for the sake of bringing the other. The conformist relationship only works between contexts that are either developed by one team of developers, or by two teams, that are able to cooperate really well with each other. When the communication issue doesn't exist and developers in both teams are careful enough to keep their colleagues informed about the changes they make in contracts they share, it might work. You might seriously consider using this type of relationships when working on smaller systems that are controlled by responsible developers with good communication skills.

User interface

So far, most of the code we wrote and most of the concepts we covered were concentrated on the server side, at what we call "the back-end". It is, however, very important to remember that our users have very little interest in the beauty or ugliness of the server-side code and all the infrastructure that supports the system they use, as soon as the system works. What they see is the screen and your system is mainly judged by its user interface. Of course, there are quite a few exceptions from this rule. Software systems for airplanes, cars, and ships, for example, rarely have any user interface at all. Only a small part of the software in the financial sector is user-facing and there are a lot of back-end systems that take care that financial transactions are executed smoothly and reliably.

And still, most of the developers create systems that have people, not machines, as main users. So, let's have a word about user-interface aspects of Domain-Driven Design in general and some particular concerns that we covered in this book already, like CQRS and Event-Sourcing.

User interface across boundaries

In the previous section, we saw different types of relationship between bounded contexts. Similar principles of integration can be applied to projections that build read-models from events that belong to different aggregates.

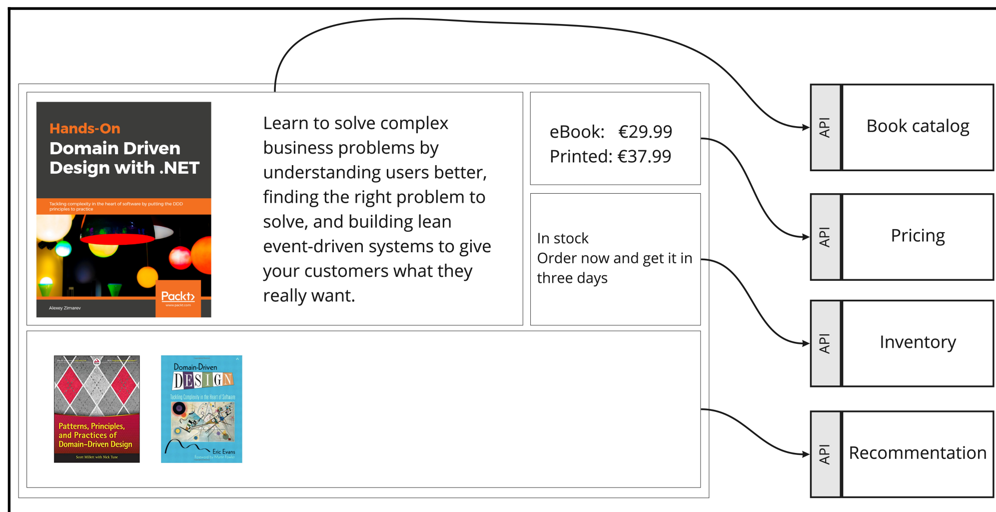
If you have a projection that only consumes events from one single context it should not be an issue, even if events come from different aggregates. The situation changes when your frontend requires a screen where the information from different contexts will be combined. There are a few ways to deal with such a requirement.

The first way follows the idea of having one single read-model per screen, regardless of where the information comes from. We can clearly see that all worries about the stability of event contracts immediately emerge. At the same time, the contract stability concern might be mitigated by identifying such read-models as a separate, "reporting" Bounded Context. In such a scenario, this new context must adopt the conformist relationship style with all components that produce events for its read-models. For a small-scale system, it might be a perfect solution. For example, when our system is hosted in a single application and all the code is located in a single repository, it might just work.

API composition

When it comes to more complex scenarios, where different contexts are owned by teams that have their own pace of development and own needs to change events, it might be hard for the reporting context to keep up with all the changes. As a result, we get coordination mess, and the team autonomy that we dreamed about is gone.

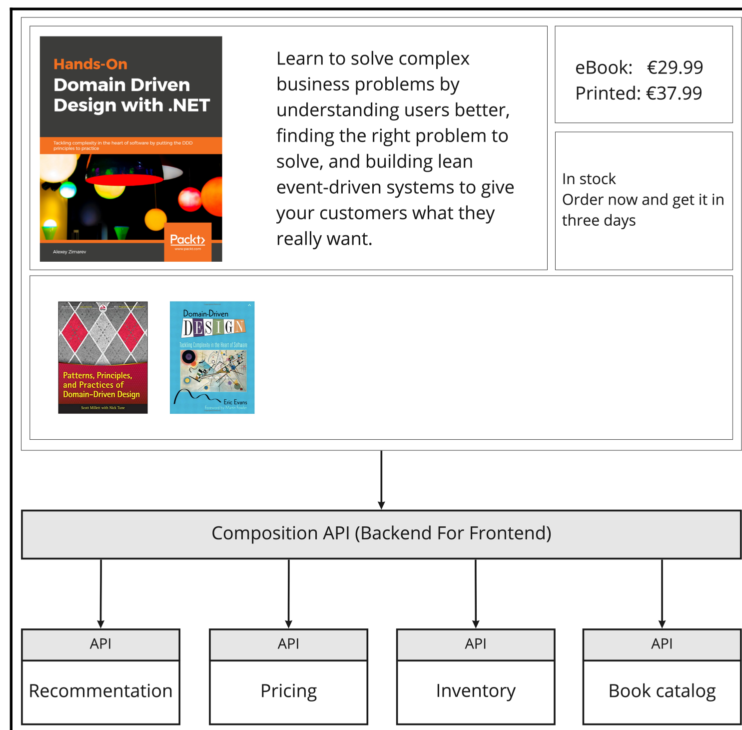
So, to mitigate dependencies between contexts, we might drop the idea to have a single read-model for a single screen, and use one of the composition patterns:



A single web page calls different APIs

Probably the most popular pattern is to use a monolithic web application that calls APIs of individual services that are provided by different Bounded Contexts. The composition then happens on the page itself. The web UI project will then work with public API contracts and won't have any unusual concerns about the stability of those contacts, since each API is stable and versioned by definition. This approach, however, has one big flaw. It usually leads to a separation between frontend and backend developers. We must always keep in mind that the web page that displays the book details on the preceding diagram is dealing with the same domain concerns as the API it talks to. When these components are being developed by people working in different teams, their collaboration will rarely be complete. In case of issues, the blame game it's the back/frontend issue has a huge chance to start.

The pattern I just described also closely resembles the **Backend for Frontend** (**BFF**) pattern, when the monolithic UI is supported by a monolithic API that serves as a proxy, conversion, and consolidation layer. The only difference is that, when you use the BFF pattern, you might perform the composition on the API side and deliver the data for the whole page in a single API call result:



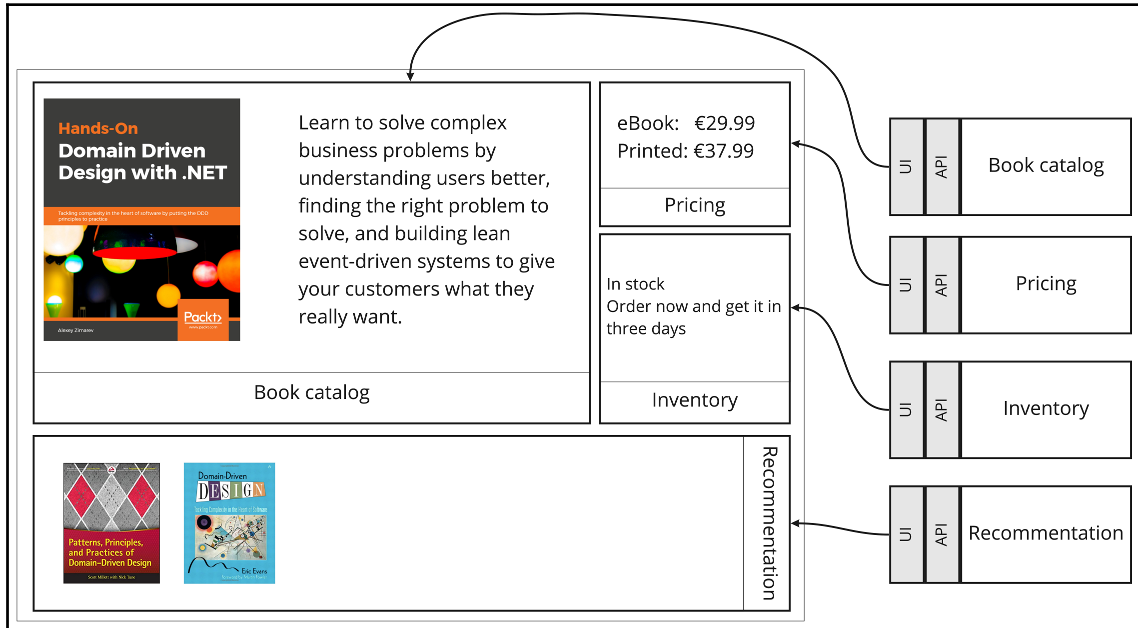
Backend for Frontend (BFF)

In my experience, teams that start adopting DDD, usually concentrate on identifying a different kind of boundaries, concentrate solely on the back-end side. Very often we see a separation between teams based on their specialization - a "front-end team" and a "back-end" team (or teams) is a somewhat stereotypical setup. Even if we get back-end teams properly aligned with business domains, the front-end team is often left alone, forced to handle all kinds of business concerns. They might have a couple of back-end developers that take care of building the BFF API. By now, you could imagine that people in that team would constantly struggle with context switching and conflicting terminology, because all their work is done in an m0nolithic API and UI. The most obvious way out of this mess for such teams would be to modularize their side of the system, just as I described in this chapter. Them they at least can split the work between different team members being more specialized in different business domains to mitigate the context switching issue. But, there will be some struggle still, because of the separation between two integral parts of the same system (the UI and the backend) is not going anywhere. I once experienced working in a company where the front-end and back-end teams got separated so much that after a couple of years of struggle with conflicts and coordination, the front-end team eventually developed their own data model and built everything on top of it. They gave up hope and for them, the struggle of keeping two models in sync, which in itself is a very hard task to solve if you don't have an event-driven system, was more beneficial than endless coordination effort. Therefore, I'd like to stress the importance of so-called vertical slices of the system, following context boundaries on all levels, including the UI. The next section will give you some ideas on how to achieve that.

Frontend composition

If we want to bring together developers that are working on one Bounded Context, regardless of their specialization, into one team, we need to find another pattern that will allow them to build a system component to provide both the backend and frontend for the Bounded Context. When it happens, we can, again, use read-models that are specifically built to satisfy the needs of the UI, but there are nuances there too.

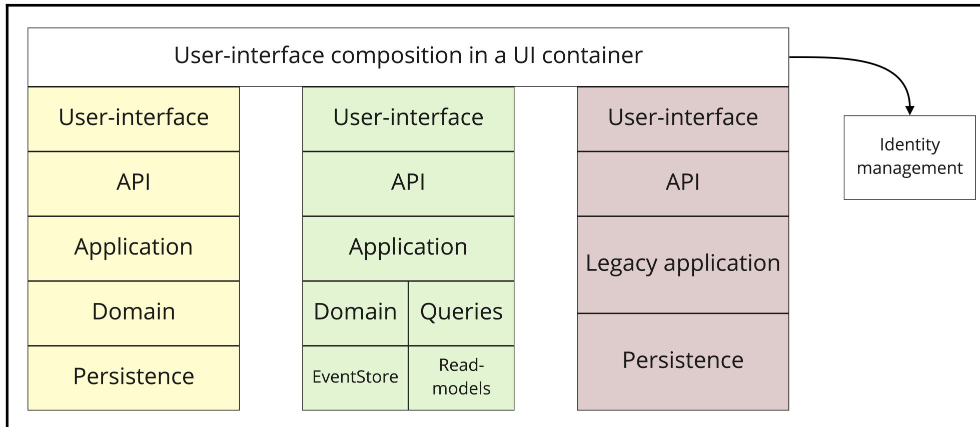
The first approach we can use is to compose complex pages with UI components that are provided by individual services. Let's use the page from the previous section as an example:



Composition of UI components

When you compose components that are provided by each service and are being hosted by those services, the web application that hosts those components becomes a simple container. It certainly needs to take care of generic concerns such as authentication. But authentication can become the concern of each component and its service. When that happens, each service is able to decide its own authorization model and visualize the data accordingly.

When a single service decides to make changes and then bring those changes to production, the team responsible for the service can do it independently, because there are no dependencies for them to manage and they just need to make sure that the new version of their UI component renders correctly in the application container.



The system is split into full-stack slices

The diagram above demonstrates the technique that I just described. You can even use parts of the legacy system and gradually move out some of its parts, as long as you have a generic place to host both new components and parts of the old system. The front-end application container can handle some generic concerns, like identity management - sign-up, login, authentication and so on. By doing that you can make the user identity accessible by all components using a session or authentication tokens.

It is also possible to use all or some aforementioned techniques in combination. For example, if you have a system that already implements the BFF pattern, you might start moving part of the page to a component, but leave the rest of the page intact. Another useful option is to embed UI components provided by independent services into the legacy web application. If you are satisfied with the result, you can work further to replace more parts of the legacy web page with new components, until it becomes just a container that hosts several components. At that moment, the legacy web application can be safely sunset.

You can imagine that my preferred method is to host the UI as close as possible to the backend service to keep all the work within the context boundary. Read-models allow us to build specific data models to provide just enough information for the UI components and we then host those components inside some sort of a container web page. I would still prefer to have one page to show data from a single context, but, when it's not feasible, I would compose several independent components on one page.

There are several ways to assemble a page with multiple components. The simplest one is to use an iframe, and this method is often used in legacy applications. You can also use `div` tags and read the inner HTML from a service using JavaScript code. Probably the best solution for the UI composition is to use web components. This technology is rather new, but all modern browsers support web components and my colleagues already have built several independent services that deliver web components and then have embedded them into a number of pages in the legacy web application.

Microservices

In the final section of the book, I'd like to touch upon the important topic of microservices. I already mentioned microservices quite a few times in this chapter, and now, it is time to elaborate on the role of microservices in the DDD world.

You can go to Google and find a lot of heated debates about whether we need to use microservices when building systems with DDD in mind or not. Another area of debate is how to match microservices with different boundaries that DDD has to offer.

I already mentioned some concerns that you would need to add to the checklist to evaluate the pros and cons of distributed systems. During the last several years, I was involved in several projects where we built distributed systems, and, from that experience, I can share some bits of advice.

When not to distribute

Before you consider splitting the system into a set of independent components, you need to know what awaits you on this path.

If you are starting a greenfield project with a relatively small scope, with a small group of developers, and you don't have a clear view of the system yet, then don't distribute. It is much faster to move on when all your code is in one repository. You can do many iterations and distill your knowledge about the domain and the domain model itself. When the system becomes larger and more developers get involved, you can consider splitting the system.

When there is an issue with context boundaries, don't distribute. It is very hard to move on at a reasonable speed if your boundaries are wrong. You will start noticing the need to continuously synchronize a lot of data and that your services start talking to each other. There is also the **service per entity** trap. Unfortunately, Google search results for *microservices* is full of diagrams where you will see things such as driver, car, and passenger in individual boxes that supposedly represent small services, and a lot of arrows between all those boxes. You will notice that all those lines are marked with HTTP captions, and some of the arrows are bidirectional. Continuous communication between services, where one service cannot live without the other service being online, is not a proper distributed system. I prefer the term "distributed monolith". In such a system, what was previously done by calling a method on an object is now done by doing exactly the same call, but over the network. Network calls have this nasty habit of failure because, by definition, the network is not reliable.

When working with distributed systems, it is important to remember eight fallacies of distributed computing:

- The network is reliable.
- The latency is zero.
- The bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- The transport cost is zero.
- The network is homogeneous.

None of the preceding statements is true. Things can break, and they will break, and it will happen at the least convenient moment.

If your team has zero experience with distributed systems, don't distribute until you all gain at least some experience by running a smaller scale project that could implement something that is not business-critical, something that your company can afford to have offline from time to time. After you gain more knowledge and experience with at least some of the issues of distribution, you can try a larger scale project. Don't put your skin in the game if you don't know the rules. Remember to learn the enterprise integration patterns so that you know how to tackle the needs for integration using a variety of established patterns.

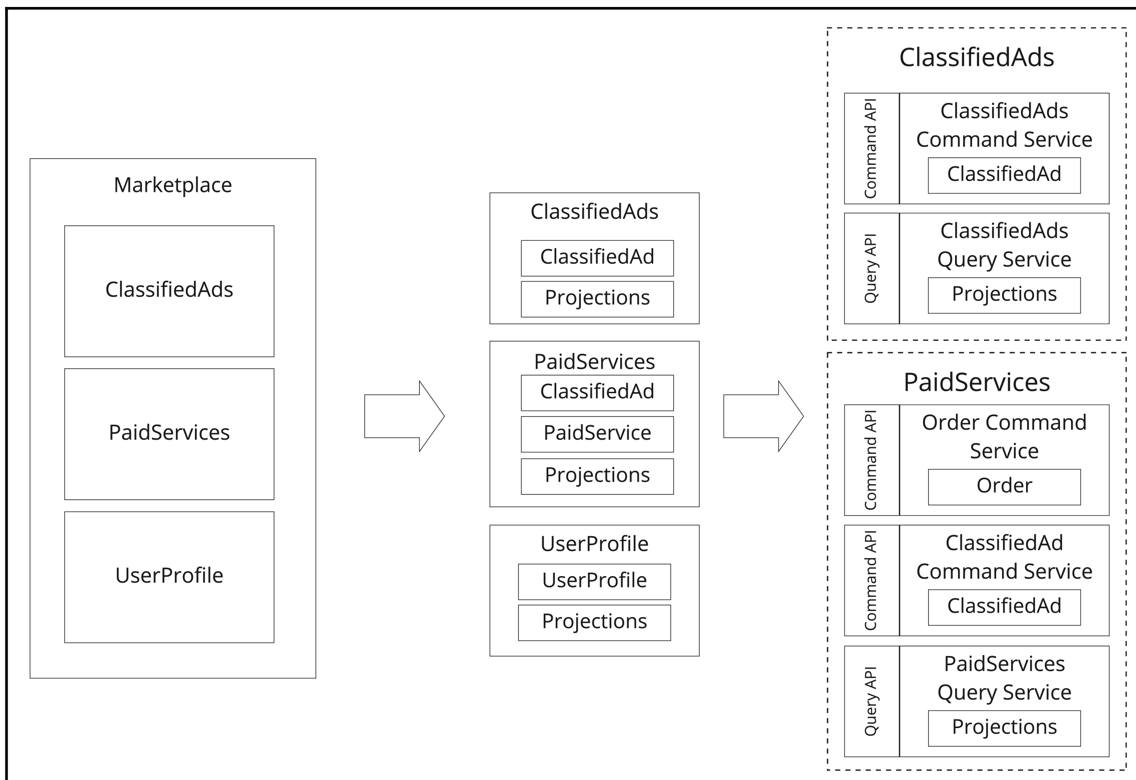
How to distribute

There are several ways to decide the scope of a microservice. Keeping DDD in mind, the first thing that comes as a suggestion is to align the microservice boundary with a single Bounded Context. You can find this suggestion in Sam Newman's best-selling book, *Building Microservice*s. During recent years, some experts, however, negated this approach as an anti-pattern. In my view, it is definitely not an anti-pattern. Matching service boundaries with Bounded Context boundaries is still a great starting point.

If you take a closer look at the sample application code for this chapter, you will find out that it can be very easily separated to several individual services. Each module is separated and modules don't reference each other. By converting the application service library to the executable project, adding the `Program.cs` file, and making minor changes in the `Startup.cs`, you will get exactly the same system running as a set of microservices. However, several important concerns aren't addressed there and I will list them in this section.

There is no single recipe to find the right level of granularity for microservices. To start with, there must be a reason to step into the chaotic world of distributed systems. Those reasons might have different natures, such as enforcing context boundaries to achieve greater autonomy for teams if nothing else works or if the code base grows too large. Other reasons may include operational concerns, such as the need for horizontal scalability, which is easier to achieve per smaller component than for the whole system. When you have a strong enough reason, you can try moving on and separate some parts of your system into microservices.

The granularity of the system greatly varies. Different Bounded Contexts have different needs for performance and scaling. Have a look at the following diagram:



Granularity level of microservices

The diagram shows different granularity levels for splitting the system and not all possibilities are covered. On the left side, we have all modules wrapping their contexts hosted in a single application. The next level could be, as I described, taking each module to a separate application and then matching the microservice size with the size of one Bounded Context. But, it is possible to move on. Aggregates also rarely reference each other directly. You can split each aggregate to a separate microservice and bring the command API with it.

One thing to remember here is to still keep an eye on the context boundaries. As I presented through the preceding diagram, those dashed boxes still represent Bounded Context as meta-services. Context boundaries are those that you need to protect the most.

Now, let's consider the projections. Normally, projections don't scale horizontally. That's because they must process events in a strict order, and, when you have several instances of the same projection consuming from the same event stream, you get competing consumers. The competing consumer's pattern is perfect for handling commands, but doesn't work for projections, because events will no longer be processed in the right order. But you can certainly move all projections for a single context together with the query API, and build a microservice. If you do that, you get more freedom to scale the command side, because command handlers are completely stateless. Possible race conditions when users make an attempt to execute multiple commands on the same aggregate are mitigated by the optimistic concurrency, and such situations happen in a single-command API instance as well, because a web API will accept multiple calls and try to process them in parallel anyway.

You can still go more granular and separate each read model and its query API to a small service. What you need to be aware of is that you will put more and more catch-up subscriptions to your Event Store and there might be some limit, although I've never seen it in real life yet.

Projections can also be scaled horizontally. All you need to do is partition your events. If you create a simple deterministic partitioning function that will generate a number from zero to ten from, for example, an aggregate ID, and put it to the event metadata, you can have one projection service instance per partition. You can even partition on the server. If you use `EventStore` to persist events, you can build a server-side projection in JavaScript that will partition all events using some deterministic algorithm and, again, has several instances of your projection matching the count of partitions.

The more advanced method, which could be useful when you reach the scale of millions of events per hour, is to have a single subscription to the Event Store and shovel all events to some high-performance event ingestion infrastructure, such as Kafka or Azure Event Hub. Both those products have partitioning built in and will reliably distribute events to several partitions without you needing to do anything except specifying the aggregate `id` as the partition key. Both Kafka and Azure Event Hub will keep all events for some time (usually from one day to a week) so, even if your projection stops for some reason, it will eventually catch up.

So, partitioning events is a reliable method to scale projections as well and you can expect the whole system to be very, very fast.

By the end, there is no single recommendation as to how to split a system into microservices, but, as you can see, the range of options is quite broad and you really need to understand both functional and operational requirements before making such an important decision. Multiple patterns can be applied in the same system where a few Bounded Contexts will happily live on one service, but some other Bounded Context will be represented as fifty microservices.

When you decide to build a distributed system, make sure the following applies to your development environment:

- A continuous integration pipeline must be set up and fully working, including the test execution.
- A continuous delivery automated pipeline is strongly recommended. All deployments must be automated.
- Things such as configuration management and infrastructure as a code help a lot, being part of the deployment pipeline.
- Consider moving away from hosting your components as Windows services. Look at Docker containers and container orchestrators, such as Kubernetes or Azure ServiceFabric.
- Establish clear diagnostics guidelines for all services. This must be a part of the "definition of done". The following concerns must be covered as a minimum:
    - Centralized logging
    - Health checks (look at ASP.NET Core Diagnostics packages)
    - Observability using metrics (check Prometheus and Grafana; there are several client libraries for .NET as well.)
    - Monitoring and alerting (Grafana has great alerting features.)

So, here, we come to the end of this book. There are a lot more things that you might want to learn about DDD and everything around it. Even after reading this book, if you haven't read the iconic book by Eric Evans, I strongly suggest that you do so. If you want to learn more and stay in the loop of all things related to DDD, seek membership in your local DDD meetup; there are quite a few around the world. If there's no DDD meetup in your area, but you know that people want to learn and teach, then start one, as I did. Also, go to a DDD conference, such as DDD Europe in Amsterdam, KanDDDinsky in Berlin or ExploreDDD in Denver. I speak at those conferences from time to time, so, if you see me, come and say "Hi!"

I wish you the best of luck in your DDD journey.

Summary

In this chapter, we got an overview and practical advice on how a system with multiple bounded contexts can be implemented in a monolithic application, using separated modules. I also explained a few options to promote those modules to independent components that can be implemented as microservices.

We also touched upon several techniques that can help to identify context boundaries, when it comes to working with user experience and user interface experts together with the backend developers in one team. The importance of the user experience in system design is frequently underestimated and it is very important to consider including all specialists in one team from the early stage of the product development, including the functional design and technical design. By doing that you would avoid many pitfalls of specialist-oriented teams that often struggle with excessive coordination and mismatch in their vision on how the product is designed, built, and deployed. Remember that bounded contexts aren't only for the backend and you can often see the separation between different business concerns on a single screen.

To conclude the chapter, I gave you several bits of advice about what to be aware of when moving to the world of distributed systems and microservices. I can't stress enough how important all aforementioned concerns are to make a microservices-oriented project a success.

The final chapter code also has some new insights into implementing different tactical Domain-Driven Design patterns using the functional approach and explained the benefits of such an approach.

Further reading

- *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans, Addison-Wesley Professional, 2003
- *Versioning in an Event Sourced System* by Greg Young, Leanpub, 2017
- *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf, Addison-Wesley Professional, 2012
- *Building Microservices: Designing Fine-Grained Systems* by Sam Newman, O'Reilly Media, 2015