8

Designing an Appointment Booking App

In previous chapters, we've seen sample implementations that were limited in scope because it would be impractical to have a full application on every covered topic.

This chapter covers the design of a barber appointment booking application, which will combine what we've learned from previous chapters:

- Dependency injection
- Unit testing
- Test doubles using mocks and fakes
- DDD
- Applying TDD

Chapter 9 and *10* will cover the implementation of this chapter. This chapter is about the business requirements and design decisions, not about the implementation (the code).

Before proceeding with this chapter and the rest of *Part 2*, I would highly recommend that you are familiar with the topics that I've listed above. They are all covered in ***Chapter 2*** to ***Chapter 7***.

In this chapter, we will cover the following:

- Business requirements to build a booking system
- The design of the system DDD-style
- The implementation routes of this system

By the end of this chapter, you will understand better a realistic DDD analysis based on a life-like problem.

Technical requirements

The code for this chapter can be found in the following GitHub repository:

https://github.com/PacktPublishing/Pragmatic-Test-Driven-Development-in-C-Sharp-and-.NET/tree/main/ch08

Collecting business requirements

You work for a software consultancy called **Unicorn Quality Solutions (UQS)**, which is implementing an appointment booking application for Heads Up Barbers, a modern barber shop with many employees.

The required application will comprise three applications:

- **Appointment booking website**: Where customers will book an appointment for hairdressing.
- **Appointment booking mobile app**: Same as the website, but a native mobile app (as opposed to a website on a mobile web browser).
- **Back office website**: This is an internal app to be used by the owner of the business. It allocates shifts for barbers (employees), cancels bookings, calculates the barbers' commission, and so on.

Phase 1 of the delivery is only the first application (booking website), which has the highest business value because it allows the users to book via desktop and on their mobile web browser.

This is our concern for the rest of Part 2 of this book. The following is a diagram showing the three phases of the project:

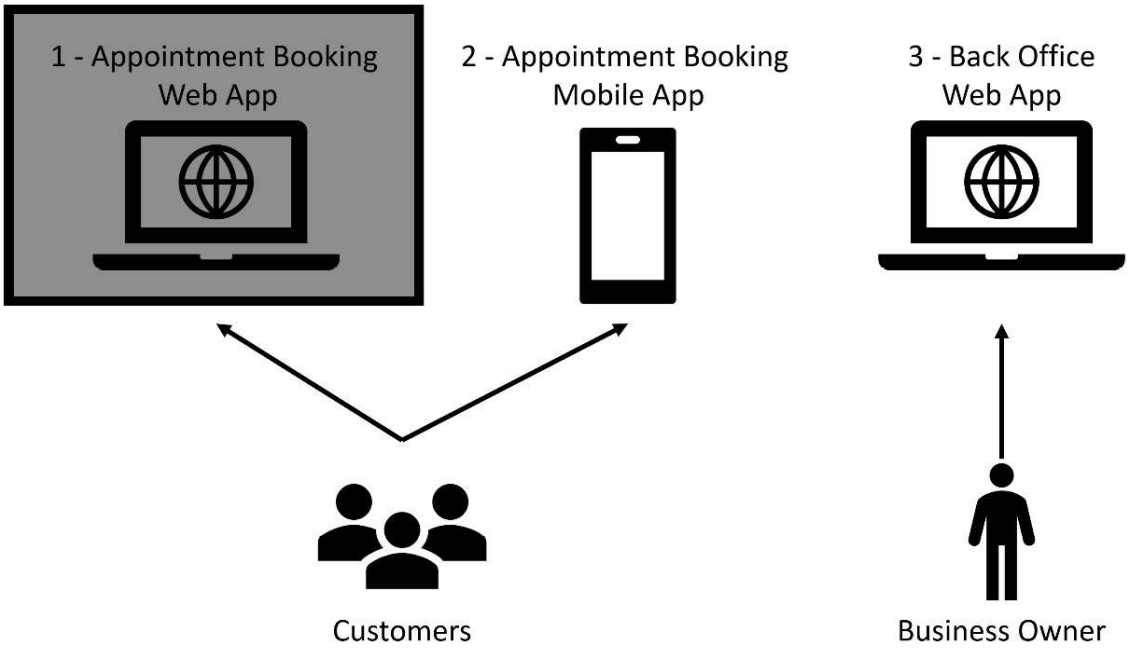

Figure 8.1 – The three required applications

Although we are only concerned with building *Phase 1*, we need to consider in our design that our architecture will include support for a mobile app at later phases.

Business goals

In this day and age, most customers like to book an appointment online, especially since COVID-19, where shops tried to reduce the concentration of people in spaces via appointments.

Heads Up Barbers wants a booking solution that aims to do the following:

- Market the available hairdressing services.
- Allow a customer to book an appointment with a specific or a random barber.
- Give barbers a rest between appointments, usually 5 minutes.
- Barbers have various shifts in the shop and they are off work on different days, so the solution should take care of picking free slots based on the availability of barbers.
- Time saving by not having to arrange appointments on the phone or in person.

Stories

After analyzing the business goals, UQS came up with more detailed requirements in the form of user stories and mockups. We will go through these next.

Story 1 – services selection

As a customer:

I want to have a list of all available services and their cost.

So I can select one for booking.

And be transferred to the booking page.

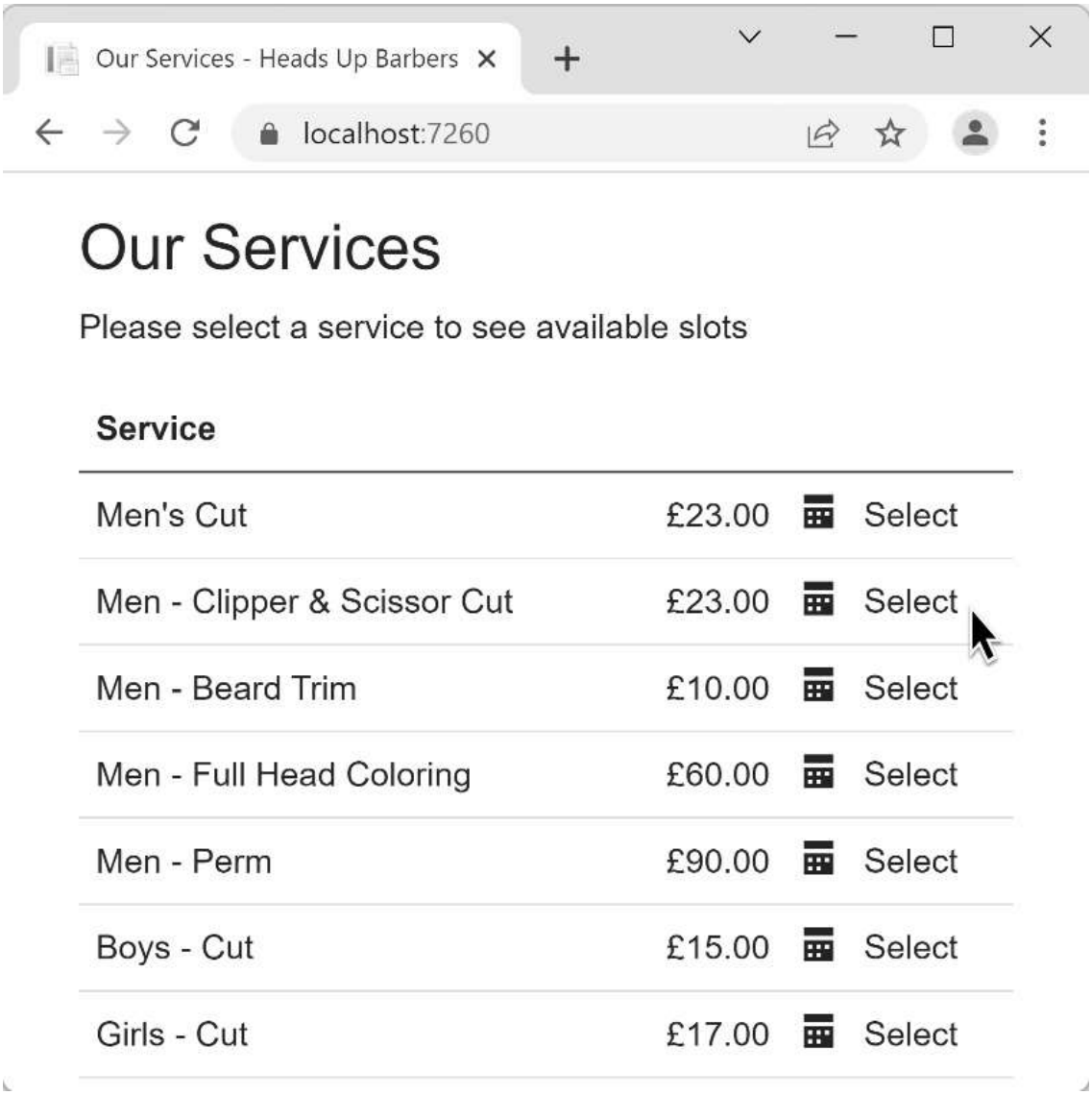

Figure 8.2 – The list of the available services and their prices

This mockup displays all the available services with their prices and **Select** hyperlinks to take the user to the booking page for the selected service.

Story 2 – default options

As a customer:

I want to have a booking page with **[Any employee]** and today's date selected by default.

So I spend less time clicking and finish booking faster.

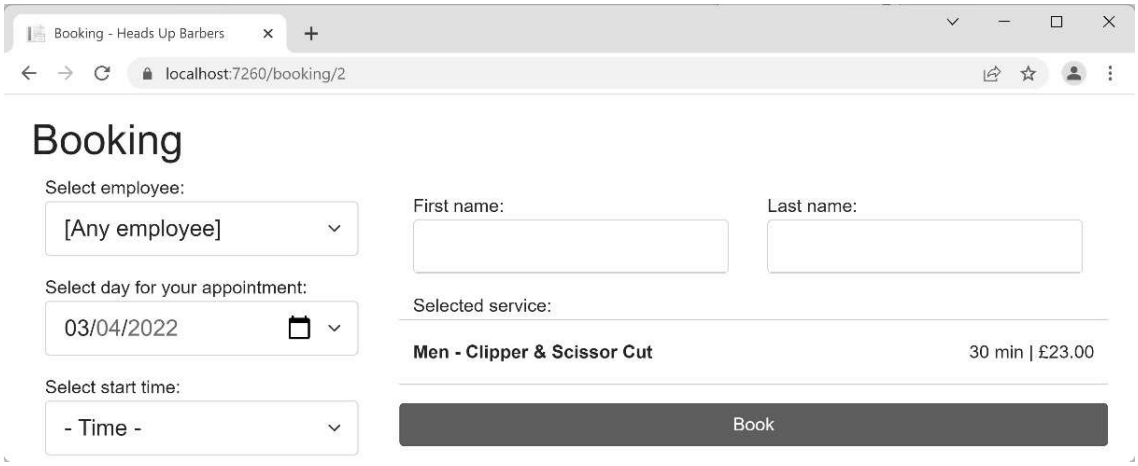

Figure 8.3 – The booking page with default options already selected

Notice that **[Any employee]** and the current day, **2022-04-03**, are selected by default.

Story 3 – select employee

As a customer:

I want to select any employee or a specific employee for my appointment.

So I can pick my favorite barber if I have one.

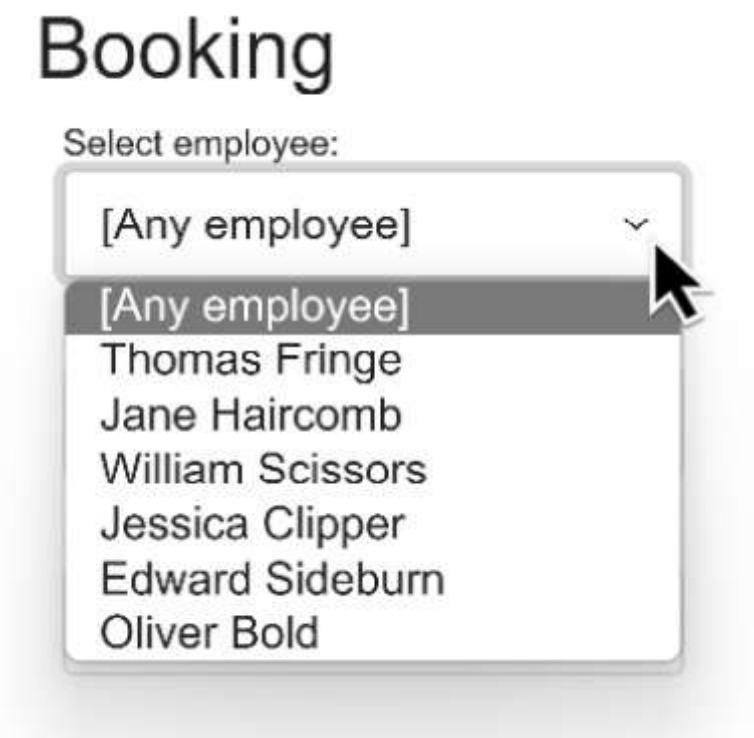

Figure 8.4 – Selecting a specific employee

The customer will have a list of barbers working for Heads Up Barbers from which they can pick their favorite one.

Story 4 – appointment days

As a business:

We want to present the customer with a 7-day window max, including the current day, to pick an appointment.

And we want to reduce this window if the selected employee is not fully available.

So we can guarantee our employees' availability for booking.

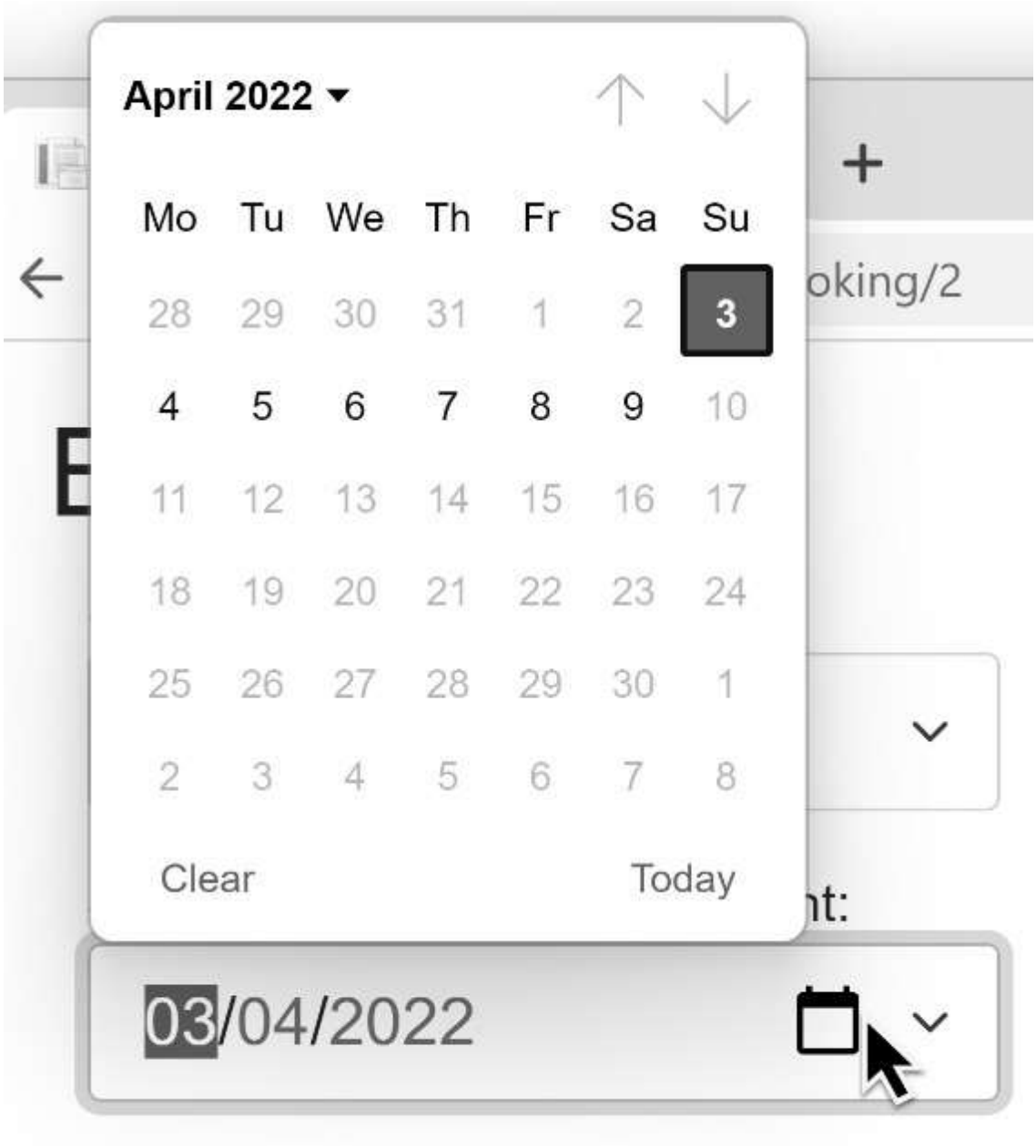

Figure 8.5 – Calendar showing a 7-day window starting 2022-04-03

The mockup will take into consideration the changes in the selected employee's schedule and show only the availability window for the selected employee.

Story 5 – time selection

As a business:

I want to present the customer with the time slots available for the selected employee for the selected date.

And take into consideration existing employee appointments and the employee's shifts.

And round up any appointment to the nearest 5 minutes.

And take into consideration the rest time of 5 minutes between appointments.

So I ensure the customer is selecting an employee that is already available.

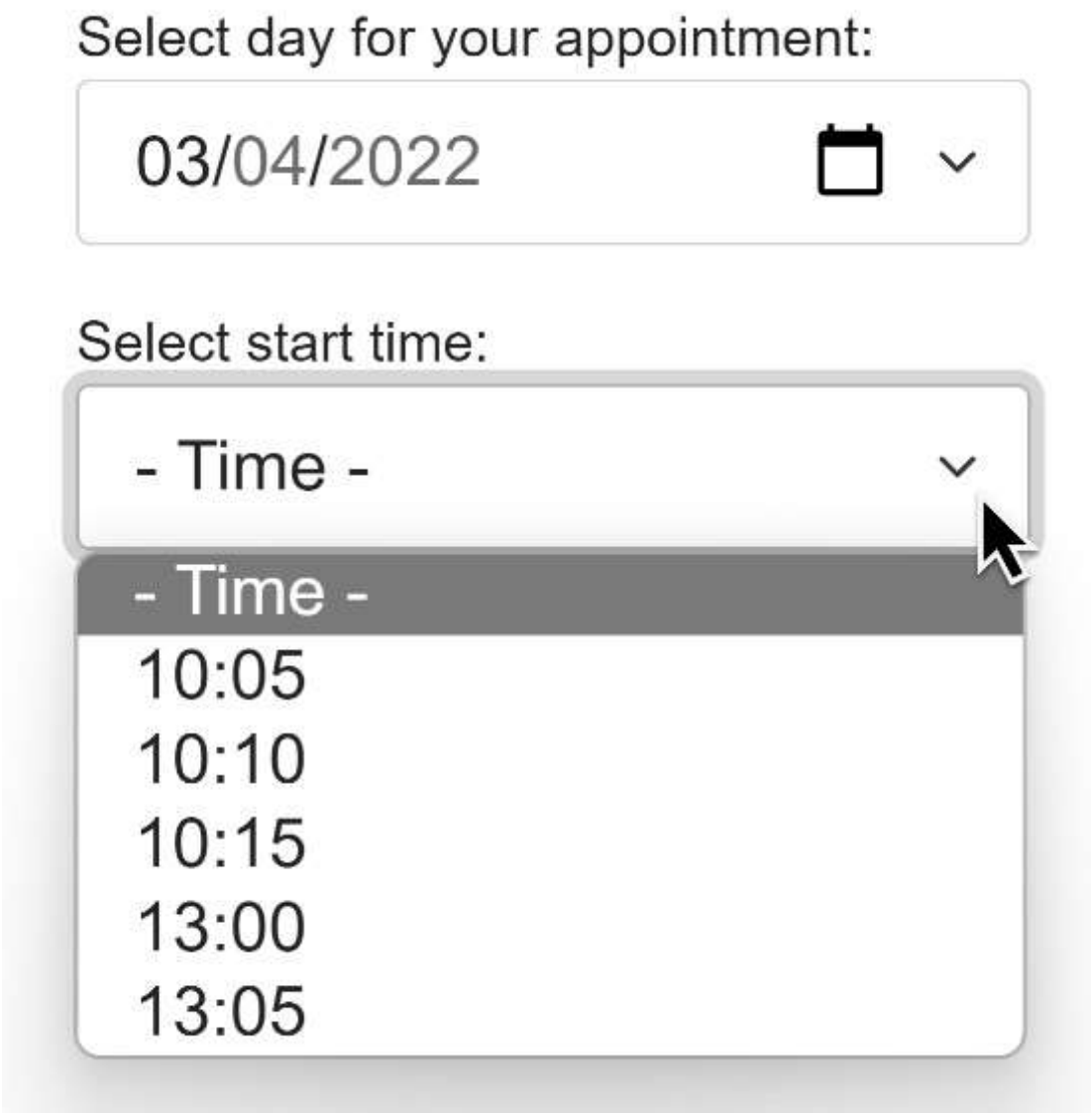

Figure 8.6 – Time slots available for the employee for the selected date

Let’s take a few examples to clarify the requirements.

Notice that all the minutes are multiples of 5.

Example 1 – no shifts are available

If an employee has no allocated shifts on the selected date, the list will be empty and the customer will be unable to book.

Example 2 – no appointments are booked

An employee, Tom, has a shift on 2022-10-03 from 9:00 to 11:10 and has no booked appointments. The customer wants to book a 30-minute-long service. The selected start time will have the following values: 09:00, 09:05, 09:10, …, 10:35, and 10:40.

Example 3 – multiple appointments booked at the end of the shift

An employee, Tom, has a shift on 2022-10-03 from 9:00 to 11:10, but he already has appointments booked from 09:35 to 11:10. The customer wants to book a 30-minute-long service. The selected start time will have the following value: 09:00. The following figure illustrates the time spans:

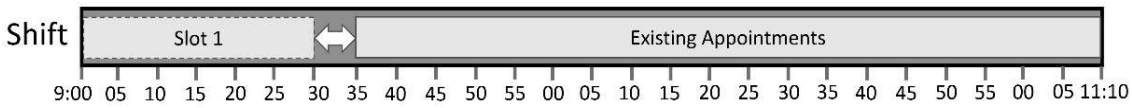

Figure 8.7 – One slot with a rest gap

Example 4 – multiple appointments booked at the end of the shift

Tom has a shift on 2022-10-03 from 9:00 to 11:10, but he already has appointments booked from 09:40 to 11:10. The customer wants to book a 30-minute-long service. The selected start time will have the following values: 09:00 and 09:05.

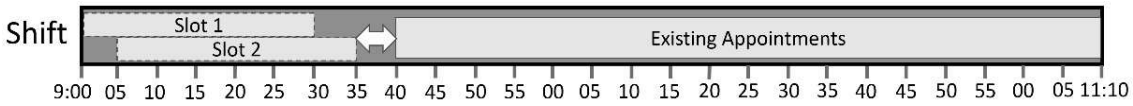

Figure 8.8 – Two slots with a rest gap

Example 5 – an appointment booked in the middle of the shift

Tom has a shift on 2022-10-03 from 9:00 to 11:10, but he already has appointments booked from 09:40 to 10:35. The customer wants to book a 30-minute-long service. The selected start time will have the following values: 09:00, 09:05, and 10:40.

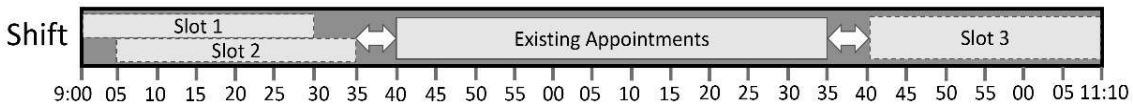

Figure 8.9 – Three slots with two rest gaps

Story 6 – name filling

As a customer:

I have to fill in my first and last name to act as my ID when I show up at the barber shop.

So I am uniquely identified.

Figure 8.10 – First name and last name fields

Story 7 – service display

As a customer:

I want a reminder of the name of the service that I picked, its price, and the required time.

So I can review my selection before hitting the **Book** button.

Story 8 – all fields are mandatory validation

As a customer:

I have to select and fill in all fields before booking.

So I won't get validation errors.

Story 9 – random selection with any employee

As a business:

When **[Any employee]** is selected.

And more than one employee is free at the selected slot.

And I hit **Book.**

A free employee is selected *randomly.*

So I ensure our employees are allocated to appointments fairly.

Example 1 – three employees are free at one slot

If the customer selects **[Any employee]** and gets three employees (Thomas, Jane, and William) who are free at 09:00, and the customer selects **09:00** and hits **Book**, Thomas, Jane, or William is allocated randomly to the appointment without taking into consideration any other factor, and one of them is selected.

Story 10 – confirmation page

As a customer:

I want to see that my appointment is booked.

So I can rest assured that it is going ahead.

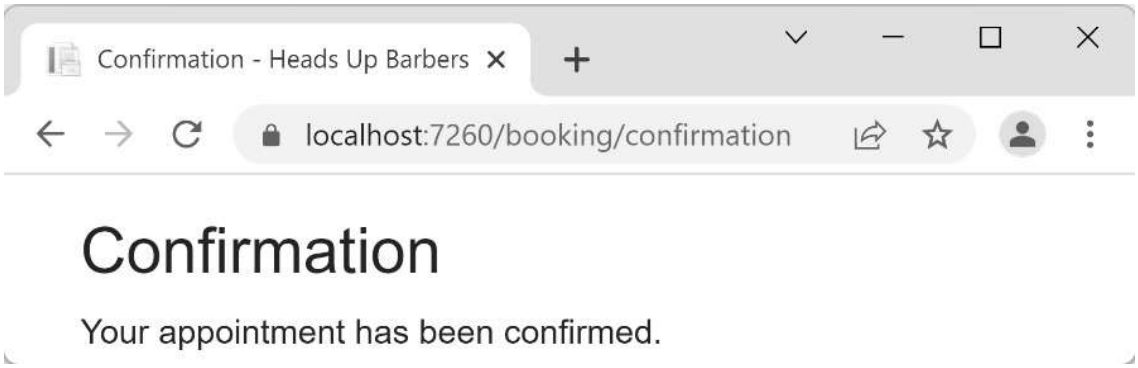

Figure 8.11 – Confirmation page

The confirmation page above is a simple static page.

You probably sensed that *Story 5* is the most demanding one from a business logic perspective, and this will be heavily targeted by our unit tests.

As you can see, the scope of the implementation is limited. In the future, we can extend this further with:

- Online payment
- User login
- Email confirmation
- And more…

However, the stories so far describe a robust life-like system. Some might call this a **minimum viable product** (**MVP**); however, I wouldn't as it might wrongly imply a lower-quality system.

Now it's time to move from the business requirements to the general guidelines for designing our system.

Designing with the DDD spirit

We have learned in the previous chapter an overview of DDD. In our implementation, we will follow the spirit of DDD to design the business classes.

Domain objects

If we were to read all the stories and think of a domain model, we might come up with the following classes:

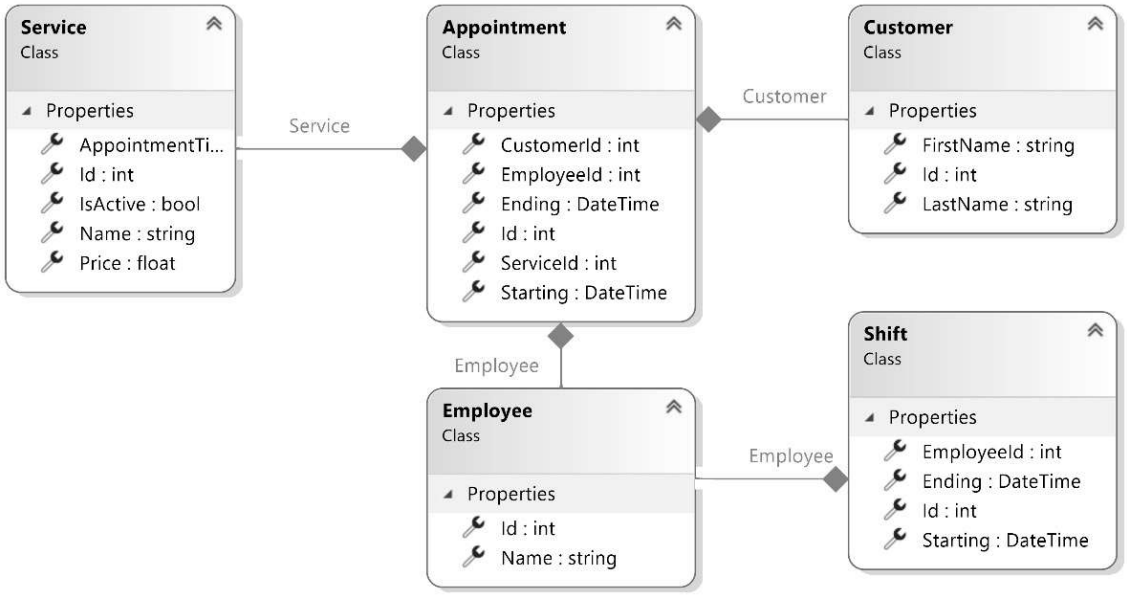

Figure 8.12 – A diagram of the domain classes

- **Service**: Represents the service offered by the barber, with `AppointmentTimeSpanInMin` being the duration of the service and `IsActive` being true to offer it to the client.
- **Customer**: Represents a customer. We are currently only interested in their name.
- **Employee**: This class will expand at a later phase to have more info, but for now, we only need the name.
- **Shift**: Represents a unique availability time for the barber. The back office application (not within scope) will allow the business owner to add shifts for employees on a daily basis to cover at least 7 days for-

ward. So, whenever we present the customer with days selection, we have at least 7 days in the future.
- **Appointment**: It is clear that an appointment links a service to an employee and a customer. It also specifies the beginning and end times of the appointment.

We have a single *aggregate* in our implementation with all the previous classes, and our *aggregate root* is clearly the `Appointment` class.

Domain services

Domain services contain the business logic that governs the system behavior. Our system will be dealing with four categories of business logic, which could lead to four domain services:

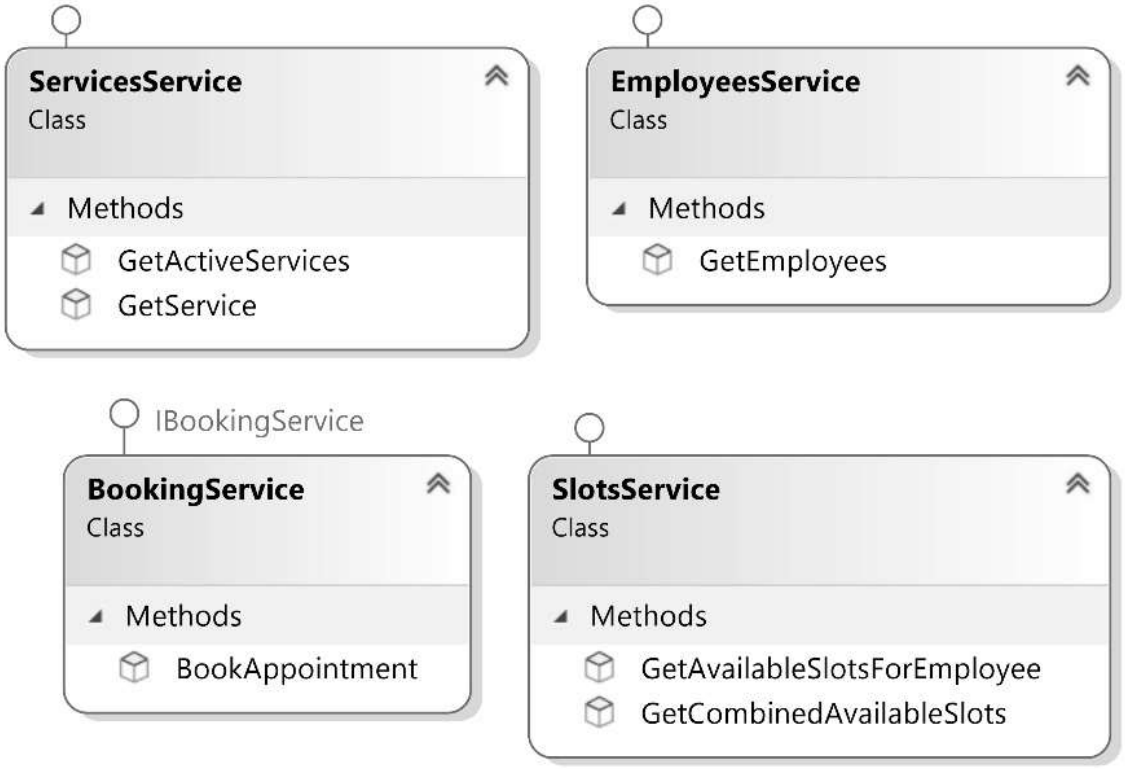

Figure 8.13 – Domain services initial design

The services at this stage are just an initial design. You usually design services driven by a TDD process rather than designing the services in advance, and this is usually done in serial, one service after the other.

System architecture

While we are only doing *Phase 1* of the system, our architecture should be ready for future phases given that a mobile app, which uses the same logic as the booking website, will be implemented in the next phase. With this in mind, the architecture in the next diagram can support all phases:

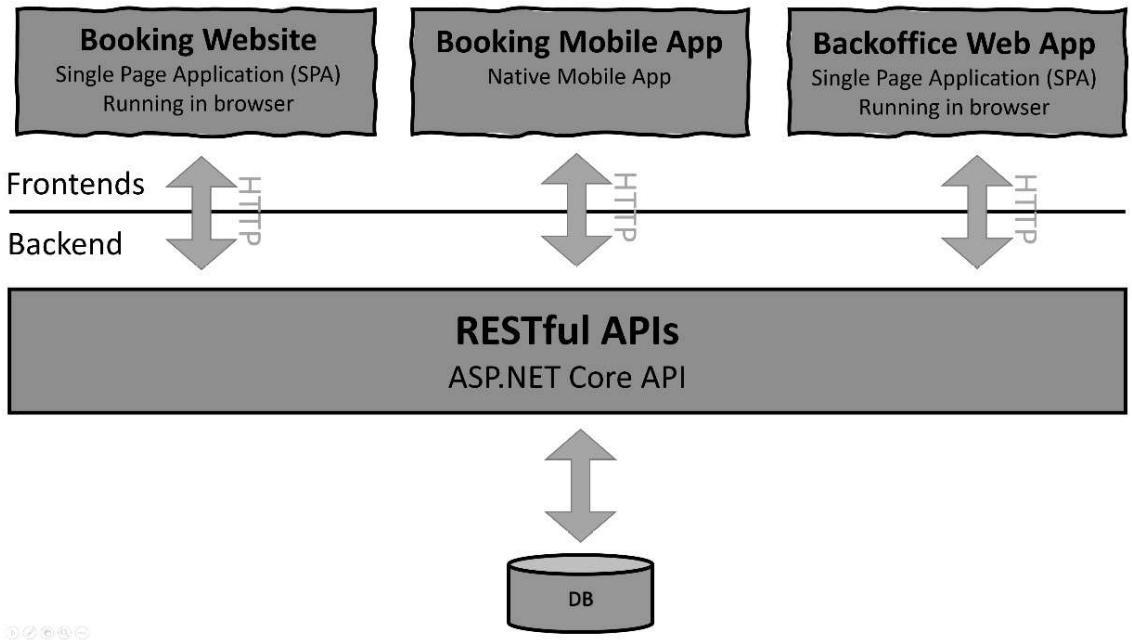

Figure 8.14 – Architecture design

Having one backend to support all the clients would embed one business logic to support all clients, so all our business logic will be behind our RESTful API application.

Also, this would make our backend act as a monolith application made up of a collection of APIs in one project and a single DB. This is alright, as this is a project with a limited scope, and going down the microservices route would be overkill.

This is a well-known architectural model where you hide the business logic behind web APIs to support multiple clients and make the logic centralized. There should be no restructuring of the architecture with the future phases when we add the booking mobile app and the back office web app.

Implementation routes

We are going to implement the backend in different ways. Each implementation will yield the same API outcome, but the point of this would be experiencing multiple unit tests and test double scenarios with each implementation.

Your team might be using one of these architectural routes, as they might be utilizing a document DB or a relational DB as in the case of most modern apps.

Frontend

In this book, we focus more on the backend, so, implementing TDD on the frontend is not covered.

IMPORTANT NOTE

There are unit testing frameworks that would test the front-end. One popular library for Blazor, which we will use here, is **bUnit,** which works side by side with xUnit.

Among all popular JavaScript **single page application** (**SPA**) platforms, such as React, Angular, and Vue, I decided to implement the frontend with Microsoft's **Blazor**.

Blazor is a web framework that relies on C# instead of JavaScript. Simply put, Blazor converts C# into a low-level language called **WebAssembly** (**Wasm**) that is understood by the browser.

I chose Blazor as I am assuming it would be easier for a C# developer without SPA experience or JavaScript/TypeScript experience.

The implementation for the frontend is minimal, and the preceding mockup screenshots in the *Stories* section are taken from the Blazor ap-

plication. You can find it in this chapter's GitHub under `Uqs.AppointmentBooking.Website`.

IMPORTANT NOTE

The implementation of this frontend is aimed at readability and minimalism, rather than web design, UX, robustness, and best practices.

To launch the website:

1. Open `UqsAppointmentBooking.sln` in VS.
2. Right-click on `Uqs.AppointmentBooking.Website` and select **Set as a Startup Project**.
3. Run from VS.

Feel free to run the website and click around. You will note that it is mocked, so it is not relying on a real DB but on sample data. The discussion about the frontend is limited to this section, as the focus of the book is TDD and the backend.

Relational database backend

Usually, using a relational DB such as SQL Server and Oracle invites **Entity Framework** (**EF**). Having your backend relying on EF has an effect on the way you organize your tests and the test double types that you are going to use.

Chapter 9, *Building an Appointment Booking App with Entity Framework and Relational DB*, will be dedicated to implementing the requirements with a relational database (SQL Server) and with EF.

Document DB backend

When using a document DB such as Cosmos DB, DynamoDB, and MongoDB, you do not use EF. That means you will be implementing more

DDD patterns such as the *Repository pattern*. This will make the implementation with a document DB fairly different than the one that uses EF from a test doubles and **dependency injection** (**DI**) point of view.

Chapter 10, *Building an App with Repositories and Document DB*, will be repeating the implementation of ***Chapter 9*** but with around 50% different code, as it will be using a document DB.

Presenting both versions will allow you to see the difference between the implementations and, hopefully, promote your understanding of test doubles and DI. However, if you are only interested in a particular type of DB, then you can choose ***Chapter 9*** or ***Chapter 10***.

The good news is that there are repetitions between these two chapters, where you will be able to spot them easily and focus on the unique implementation.

Using the Mediator pattern

When using the Mediator pattern, all your design changes, and your testing and test doubles follow suit. The Mediator pattern is a two-edged sword; it has a steep learning curve, but when learned and implemented, it provides a higher level of component separation of concern. It will also alter the structure of your unit tests. The Mediator pattern is outside the scope of this book, and it is mentioned here to point you to discover related patterns that affect your DI implementation and your unit tests.

Hopefully, by the end of *Part 2*, you've got a real sense of how to implement TDD in a more realistic setting.

Summary

We've seen fairly decent user requirements and we've seen a potential design for the system. This chapter was the beginning of *putting everything together*.

You've also seen a design based on DDD, which will change into code in later chapters. We have also discussed implementation routes that will affect the way we do testing and test doubles.

Sophisticated and modern projects use concepts from DDD. By now, after analyzing a full project, I hope that the DDD terminology will start sounding familiar and aid you in building your next project and help you communicate with expert developers.

The next chapter is an implementation of this chapter, but with a focus on SQL Server and EF.

Further reading

To learn more about the topics discussed in the chapter, you can refer to the following link:

- *Mediator NuGet popular lib in .NET*: **https://github.com/jbogard/MediatR**