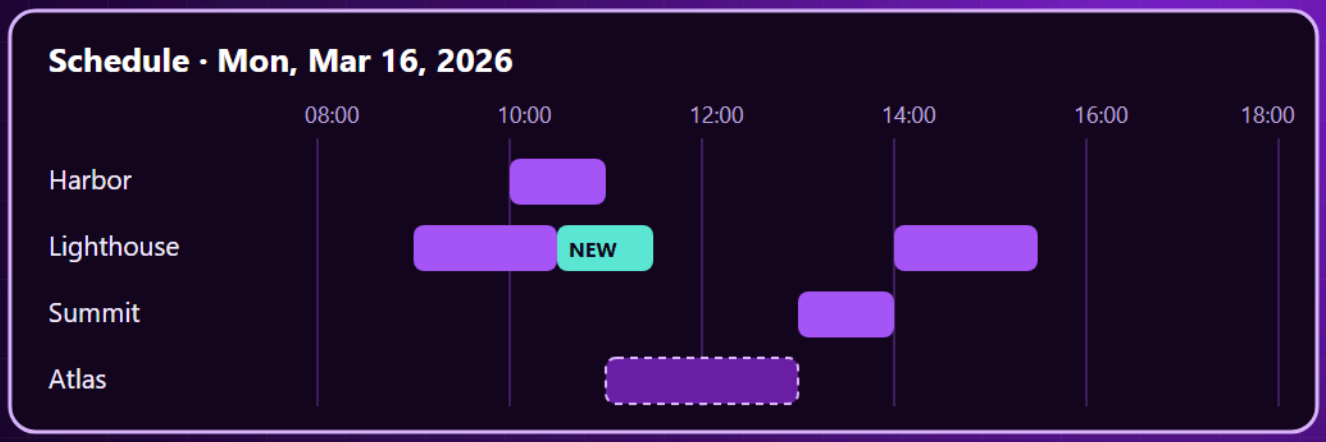



```razor
@page "/rooms"
@inject RoomDeskApiClient Api

<RoomCard Room="room"
  OnSelected="BookAsync" />

@code {
  rooms = await Api.GetRoomsAsync();
```

201 Created

```
POST /api/bookings

{ "id": 9,
  "roomName": "Lighthouse",
  "start": "10:30",
  "end": "11:30" }
```

Location: /api/bookings/9



A COMPLETE, PRACTICAL GUIDE

ASP.NET Core and Blazor Full-Stack

Build real web apps from database to browser with modern .NET

Web APIs · EF Core · Blazor · SignalR · Identity
Authorization · Testing · Docker · CI/CD · Deployment

C# · ASP.NET CORE · BLAZOR · EF CORE · XUNIT

THENEXBOOKS

ASP.NET Core and Blazor Full-Stack

Build Real Web Apps with Web APIs, EF Core, Blazor, Authentication, Testing, and Deployment in Modern .NET

From an empty folder to a secured, tested, deployed booking system — built step by step

First Edition

THENEXBOOKS

ASP.NET Core and Blazor Full-Stack
Build Real Web Apps with Web APIs, EF Core, Blazor, Authentication, Testing, and Deployment in Modern .NET

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. .NET, ASP.NET Core, Blazor, Entity Framework Core, and the libraries discussed in this book change over time; templates, package versions, default settings, and tool output can differ between releases. Always consult the official documentation for the versions you use, and never use the demo passwords or development settings shown in this book in a production system.

Microsoft, .NET, ASP.NET, Blazor, Entity Framework, Visual Studio, Visual Studio Code, Azure, SQL Server, Windows, and IIS are trademarks of the Microsoft group of companies. SQLite is in the public domain. Docker is a trademark of Docker, Inc. GitHub is a trademark of GitHub, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Microsoft Corporation, the .NET Foundation, or any other company mentioned.

Brightwave Labs and RoomDesk, the company and the meeting-room booking system used as the running example in this book, are fictional, as are their employees, rooms, bookings, and all sample data. Email addresses use the reserved `example` domain.

Preface

Most business web applications look alike from a distance. There is a database with a few related tables, a set of rules that must never be broken, an API that other systems call, a user interface that people use all day, a login screen, and a deployment that has to keep running after the developer goes home. Building any one of these pieces is not hard. Building all of them so that they fit together — consistent names, honest error messages, secure by default, tested, and easy to change — is where most projects struggle.

ASP.NET Core and Blazor let you build every layer of such an application in one language and one toolchain. You write the Web API, the data access, the business rules, the interactive UI, and the tests in C#, share the same types between server and client, and deploy to Windows, Linux, containers, or the cloud. This book shows you how to do that properly, with modern .NET and the practices that professional teams use.

We do not tour features one by one. We build one real application from an empty folder to production, and introduce each part of the platform at the moment it solves a problem we actually have. Along the way we explain *why* things work as they do, so that you can make good decisions in your own projects rather than copy recipes.

Who This Book Is For

- **Developers who know some programming** — in C#, Java, JavaScript, Python, or another language — and want to build complete web applications with .NET.
- **Backend developers** who want to add a modern, component-based user interface without switching to a JavaScript framework.
- **Front-end developers** who want to understand the server side: APIs, databases, authentication, and deployment.
- **.NET Framework, Web Forms, and MVC 5 developers** who need to move existing skills and applications to modern ASP.NET Core.

What You Will Learn

- How ASP.NET Core works: hosting, the request pipeline, middleware, dependency injection, configuration, options, and logging.
- The C# features modern web code relies on: records, nullable reference types, pattern matching, LINQ, async/await, and the newest language additions.
- How to model and query data with Entity Framework Core: relationships, constraints, migrations, seeding, efficient LINQ, transactions, and concurrency.
- How to design and build a clean Web API: resources and DTOs, status codes, validation, Problem Details, controllers and filters, OpenAPI documentation, and resilient HTTP clients.

- How to build interactive user interfaces with Blazor: components, render modes, data binding, forms and validation, state, JavaScript interop, and real-time updates with SignalR.
- How to secure the whole stack: ASP.NET Core Identity, roles, policies, resource-based authorization, cookies and tokens, CORS, antiforgery, rate limiting, and secrets.
- How to test and ship: unit tests with xUnit, integration tests with WebApplicationFactory, component tests with bUnit, caching, health checks, observability, containers, IIS, Azure, and CI/CD.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) sets up the tools, covers the C# you need, and explains hosting, middleware, dependency injection, configuration, logging, routing, and minimal APIs. **Part II — Data with EF Core** (Chapters 6–10) builds the database layer: entities, relationships, migrations, queries, business rules, transactions, and concurrency. **Part III — Web APIs** (Chapters 11–15) turns that into a complete, documented API with validation, error handling, controllers, OpenAPI, and resilient clients. **Part IV — Blazor** (Chapters 16–20) builds the user interface: components, binding, forms, the booking screens, state, JavaScript interop, and real-time updates. **Part V — Security** (Chapters 21–24) adds authentication, authorization, a secure connection between the Blazor app and the API, and essential web security. **Part VI — Testing and Production** (Chapters 25–28) covers unit, integration, and component testing, performance, caching, observability, and deployment. The appendices provide a cheat sheet, templates and reference material, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: RoomDesk

Throughout the book we build **RoomDesk**, the meeting-room booking system of **Brightwave Labs**, a fictional product-design and software consultancy. Brightwave Labs has one office with six meeting rooms on three floors, from the two-person Focus Pod to the twenty-seat Atlas boardroom. Employees need to see which rooms are free, book a room for a meeting, and cancel when plans change; the office manager needs to add, edit, and retire rooms. The rules are simple to state and easy to get wrong: no two bookings may overlap in the same room, meetings happen on weekdays between 08:00 and 18:00 in 15-minute steps, a meeting must fit the room, and nobody can book more than 60 days ahead.

RoomDesk starts as a single endpoint returning a list of rooms. It grows into a solution with a shared core library, a Web API backed by Entity Framework Core and SQLite, a Blazor Web App with a live schedule, ASP.NET Core Identity with roles and resource-based rules, a full test suite, and a deployment pipeline. The sample data is small enough to check by hand, and every API response, error message, and test result in this book was produced by running the code.

Versions and Conventions

The examples target **.NET 10**, the long-term support release, with **C# 14** and **EF Core 10**. Most of the code also works on .NET 8; where it does not, a Note explains the difference. Newer releases may change templates, defaults, and package versions, so if something on your screen differs slightly from the book, check the release notes for your version. The book uses the .NET command-line interface (`dotnet`) so that every step works the same on Windows, macOS, and Linux, in any editor.

RoomDesk works in the office's local time. To make every example reproducible, the development configuration freezes the application's clock at **Monday, March 16, 2026, 07:30** — Chapter 4 shows how. Dates and times in the book use US English formatting, such as `Mon, Mar 16, 2026` and 24-hour times such as `09:00`.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause bugs, data loss, or security problems.

Commands are shown with a `$` prompt, such as `$ dotnet run`. File names appear above code listings, with paths relative to the solution folder, such as `src/RoomDesk.Api/Program.cs`. Long lines are wrapped to fit the page, and `// ...` marks code we left out because it has not changed.

Contents

Preface

PART I — FOUNDATIONS

1 ASP.NET Core and Blazor: The Platform, the Tools, and Your First App

2 C# for Web Developers: The Modern Language Features You Will Use Every Day

3 How ASP.NET Core Works: Hosting, the Request Pipeline, and Middleware

4 Dependency Injection, Configuration, Options, and Logging

5 Routing and Minimal APIs: Your First RoomDesk Endpoints

PART II — DATA WITH EF CORE

6 EF Core Fundamentals: DbContext, Entities, and SQLite

7 Modeling Relationships, Constraints, and Configuration

8 Migrations, Seeding, and Evolving the Schema

9 Querying Well: LINQ, Loading Strategies, Projections, and Reports

10 Business Rules, Transactions, and Concurrency

PART III — WEB APIS

11 Designing the Booking API: Resources, DTOs, and HTTP Semantics

12 Validation, Errors, and Problem Details

13 Controllers, Filters, and Organizing a Growing API

14 OpenAPI Documentation, API Explorers, and Versioning

15 Calling APIs: HttpClientFactory, Typed Clients, and Resilience

PART IV — BLAZOR

16 Blazor Fundamentals: Components, Razor Syntax, and Render Modes

17 Data Binding, Events, Lifecycle, and Component Communication

18 Forms and Validation in Blazor

19 Building the Booking UI: Rooms, Schedule, and the Booking Flow

20 State, JavaScript Interop, and Real-Time Updates with SignalR

PART V — SECURITY

21 Authentication with ASP.NET Core Identity

22 Authorization: Roles, Policies, and Resource-Based Rules

23 Securing the Blazor App and the API Together

24 Web Security Essentials: HTTPS, CORS, CSRF, Rate Limiting, and Secrets

PART VI — TESTING AND PRODUCTION

25 Unit Testing with xUnit: Rules, Services, and Test Doubles

26 Integration and Component Testing: WebApplicationFactory and bUnit

27 Performance, Caching, Health Checks, and Observability

28 Deployment and Case Study: Publishing, Containers, IIS, Azure, and CI/CD

APPENDICES

A ASP.NET Core and Blazor Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

The platform, the tools, and the C# you need; how an ASP.NET Core application starts and handles a request; middleware, dependency injection, configuration, options, and logging; and the first RoomDesk endpoints built with minimal APIs.

1 · The Platform, the Tools, and Your First App

2 · C# for Web Developers

3 · Hosting, the Request Pipeline, and Middleware

4 · Dependency Injection, Configuration, and Logging

5 · Routing and Minimal APIs

CHAPTER 1 · PART I — FOUNDATIONS

ASP.NET Core and Blazor: The Platform, the Tools, and Your First App

Before we write a single business rule, we need a clear picture of the platform and a working solution on disk. In this chapter you install the .NET 10 SDK, learn what ASP.NET Core and Blazor actually are, meet Brightwave Labs and its booking problem, and create the RoomDesk solution that the rest of the book grows into a production system.

What ASP.NET Core Is

ASP.NET Core is Microsoft's open-source framework for building web applications and services on .NET. It is the successor to the classic ASP.NET that shipped with the Windows-only .NET Framework, but it is not an upgrade of that code base: it was rewritten from the ground up to be modular, fast to start, and cross-platform. An ASP.NET Core application runs on Windows, Linux, and macOS, inside a container, behind IIS, or on its own as a plain process.

At the heart of every ASP.NET Core application sits **Kestrel**, the built-in, cross-platform HTTP server. When you run an app with `dotnet run`, Kestrel opens a port and starts accepting requests. You do not install a separate web server to develop or even to run in production. In many deployments Kestrel sits behind a reverse proxy such as Nginx, YARP, or IIS (through the ASP.NET Core Module), which handles concerns like TLS termination or sharing one port between several sites, but the application itself is always the same Kestrel-based process.

On top of Kestrel, ASP.NET Core gives you a set of building blocks that you combine as needed:

- **Hosting and the request pipeline**: a small program that configures services, then a chain of middleware components that every HTTP request flows through (Chapter 3).
- **Dependency injection, configuration, options, and logging**, built in rather than bolted on (Chapter 4).
- **Minimal APIs and MVC controllers** for HTTP APIs that return JSON (Chapters 5 and 11–13).
- **Razor Pages and MVC views** for classic server-rendered HTML. We do not use them in this book, but they remain fully supported.
- **Blazor** for component-based user interfaces written in C# (Part IV).
- **SignalR** for real-time communication over WebSockets and fallbacks (Chapter 20).
- **Authentication, authorization, data protection, CORS, rate limiting**, and the other security features every real application needs (Part V).

Data access is not part of ASP.NET Core itself. Most .NET teams use **Entity Framework Core** (EF Core), the object-relational mapper that ships as a separate set of NuGet packages, and that is what RoomDesk uses from Chapter 6 onward.

What Blazor Is

Blazor is the part of ASP.NET Core that lets you build web user interfaces from **components** written in C# and Razor markup instead of JavaScript. A component is a reusable piece of UI, such as a room card, a date picker, or a whole page, with its own parameters, state, and event handlers. Components render HTML; the interesting question is *where* the C# code runs and *when* the page becomes interactive. Blazor offers several answers, called render modes:

- **Static server-side rendering (static SSR)**: the component runs on the server, produces HTML, and the browser receives an ordinary page with no live connection. This is the default in a modern Blazor Web App.
- **Interactive Server**: the component runs on the server and stays alive; browser events travel to the server over a persistent SignalR connection, and only UI changes travel back.
- **Interactive WebAssembly**: the component, the .NET runtime, and your assemblies are downloaded to the browser and run there, on WebAssembly.
- **Interactive Auto**: starts with Server for a fast first interaction, then switches to WebAssembly once the runtime has downloaded in the background.

You can mix these modes in one application, even per page or per component. RoomDesk uses static SSR for simple pages such as the login form and Interactive Server for the screens that react to clicks, like the schedule grid. Chapter 16 explains the trade-offs properly; for now it is enough to know that the choice exists and that we made it deliberately.

What "Full-Stack .NET" Means

A full-stack web application has several layers: a database, data access code, business rules, an HTTP API, a user interface, authentication, tests, and a way to deploy all of it. In a typical JavaScript-front-end project those layers are split between two languages, two package ecosystems, two build systems, and often two teams. With ASP.NET Core and Blazor, every layer can be written in C# and built with one command.

The most tangible benefit is **shared code**. In RoomDesk, the class library `RoomDesk.Core` contains the entities, the booking rules, and the data transfer objects (DTOs). The API uses those DTOs to send JSON, and the Blazor app uses *the same types* to read it. Rename a property and the compiler tells you about every place on both sides that needs to change. The Blazor form validates with the same DataAnnotations attributes, and the same limits, that the API applies to `CreateBookingRequest`. There is no hand-maintained TypeScript copy of the model that drifts out of date.

You also get one debugger, one test framework, and one set of dependency updates for the whole stack.

When a JavaScript front end is the better choice

It would be dishonest to claim Blazor is always the right answer. Choose a JavaScript framework such as React, Angular, or Vue when your team is already productive with it and has no C# experience, when you depend heavily on JavaScript-only UI libraries, when you need a public, search-engine-heavy site where every kilobyte of download matters and you already have a front-end pipeline you trust, or when the front end is a separate product with its own release cycle and team. ASP.NET Core is an excellent back end for those front ends too; Parts II, III, and V of this book apply unchanged.

Blazor shines in line-of-business applications like RoomDesk: forms, grids, dashboards, internal tools, and portals built by teams who already write C# on the server. That is a very large share of the web applications companies actually build.

> **NOTE**
>
> If you come from ASP.NET MVC 5 or Web Forms on the .NET Framework, many ideas carry over (controllers, Razor syntax, model binding, filters), but the hosting model is new: there is no `Global.asax`, no `web.config`-driven pipeline, and no dependency on IIS. Chapter 3 explains the replacement, and the differences are pointed out where they matter.

.NET Versions and Support

.NET ships a new major version every November. Even-numbered releases are **long-term support (LTS)** releases with three years of support; odd-numbered releases are **standard-term support (STS)** releases with a shorter window. Check the official .NET support policy for exact end-of-support dates, because they are what your operations team plans around.

Version	Released	Support type	Role in this book
.NET 8	November 2023	LTS	Still common; differences noted
.NET 9	November 2024	STS	Mentioned only when useful
.NET 10	November 2025	LTS	The baseline for all code
.NET 11	Planned November 2026	STS	In preview; not used

Everything in this book targets **.NET 10 with C# 14** and builds with the 10.0 SDK. At the time of writing, .NET 11 is available only as a **preview**; previews are great for experiments but are not supported for production, so we do not use them. Because many teams still run .NET 8, you will find Notes wherever .NET 8 behaves differently in a way that affects RoomDesk, such as the solution file format later in this chapter or the built-in validation in Chapter 12.

Two version numbers are easy to confuse. The **runtime** version (for example 10.0.12) is what runs your application. The **SDK** version (for example 10.0.401) is the toolset that builds it; SDK "feature bands" such as 10.0.1xx, 10.0.2xx, and 10.0.4xx add tooling improvements while targeting the same .NET 10 runtime. Your machine can hold many SDKs and runtimes side by side.

Installing the .NET SDK

You need the **.NET 10 SDK**, which includes the runtimes for console apps and ASP.NET Core, the compiler, the `dotnet` command-line interface (CLI), and all project templates used in this book. Download it from the official .NET website (`dot.net`), choosing the SDK rather than a runtime-only package.

- **Windows:** run the SDK installer, or use the Windows Package Manager. Run `winget search Microsoft.DotNet.SDK` to find the exact package ID for the 10.0 SDK, then `winget install` it. Installing a recent Visual Studio with the ASP.NET workload also installs an SDK.
- **macOS:** run the installer package for your processor (Arm64 for Apple silicon, x64 for Intel).
- **Linux:** many distributions ship .NET in their own package feeds (for example `dotnet-sdk-10.0` on recent Ubuntu releases); Microsoft also publishes packages and a `dotnet-install` script for distributions without one. Follow the instructions for your distribution on the .NET website.

Open a new terminal after installing, so that the updated `PATH` is picked up, and check what you have:

Terminal

```
$ dotnet --version
10.0.401

$ dotnet --list-sdks
8.0.302 [C:\Program Files\dotnet\sdk]
9.0.317 [C:\Program Files\dotnet\sdk]
10.0.401 [C:\Program Files\dotnet\sdk]
```

Your list will differ; what matters is that a 10.0 SDK is present. `dotnet --version` prints the SDK that the CLI will use in the current folder, which is normally the newest one installed. For a full report, including the operating system and every runtime, run `dotnet --info`:

Terminal

```
$ dotnet --info
.NET SDK:
 Version:           10.0.401
 Commit:            e34a38d2ae
 Workload version:  10.0.400-manifests.a3b2712e
 MSBuild version:   18.9.11+e34a38d2a

Runtime Environment:
 OS Name:     Windows
 OS Version:  10.0.22631
 OS Platform: Windows
```

```
 RID:         win-x64
 Base Path:   C:\Program Files\dotnet\sdk\10.0.401\
...
```

Further down, the same command lists the installed runtimes. You should see `Microsoft.AspNetCore.App 10.0.x` and `Microsoft.NETCore.App 10.0.x`; those are the shared frameworks your RoomDesk projects will run on. You can also print just that part with `dotnet --list-runtimes`.

> **TIP**
>
> On a team, pin the SDK with a `global.json` file in the repository root so everyone builds with the same feature band. `dotnet new globaljson --sdk-version 10.0.401 --roll-forward latestFeature` creates one that accepts 10.0.401 or any newer 10.0 SDK. Without the file, the newest installed SDK is used, which is usually what you want on a personal machine.

Choosing an editor

Every step in this book uses the `dotnet` CLI, so any editor works. The three most common choices are a recent **Visual Studio** (Visual Studio 2026 or Visual Studio 2022 with current updates) on Windows, **Visual Studio Code** with the **C# Dev Kit** extension on any operating system, and **JetBrains Rider** on any operating system. All three understand `.slnx` solution files, Razor components, and `.http` request files in their current versions; if yours does not, update it. Use the one you enjoy. When an editor feature saves real time, such as running a single test from the gutter, we mention it, but you never need it to follow along.

The Template Catalog

The SDK installs project templates that generate a ready-to-build starting point. To see the ones related to the web, filter the list:

Terminal (excerpt)

```
$ dotnet new list web
These templates matched your input: 'web'

Template Name                                 Short Name        Language  Tags
--------------------------------------------  ----------------  --------  ------------
ASP.NET Core Empty                            web               [C#],F#   Web/Empty
ASP.NET Core Web API                          webapi            [C#],F#   Web/WebAPI/...
ASP.NET Core Web API (native AOT)             webapiaot         [C#]      Web/Web API/...
ASP.NET Core Web App (Model-View-Controller)  mvc               [C#],F#   Web/MVC
ASP.NET Core Web App (Razor Pages)            webapp,razor      [C#]      Web/MVC/...
Blazor Web App                                blazor            [C#]      Web/Blazor/...
Blazor WebAssembly Standalone App             blazorwasm        [C#]      Web/Blazor/...
...
```

We use four templates for RoomDesk: `classlib` for the shared core library, `webapi` for the HTTP API, `blazor` for the Blazor Web App, and `xunit` for tests. Each template has options; `dotnet new webapi --help` lists them. A few that come up in this book: `--use-controllers` (generate a controller instead of minimal APIs), `--no-openapi`, and `--auth` for the API; `--interactivity` (`None`, `Server`, `WebAssembly`, or `Auto`), `--all-interactive`, and `--empty` for Blazor.

Meet Brightwave Labs and RoomDesk

Our running example belongs to **Brightwave Labs**, a fictional product-design and software consultancy. It has one office, **Brightwave Labs HQ**, with three floors and six meeting rooms. Today, people book rooms on a shared spreadsheet, and double bookings are a weekly source of friction. The office manager, **Ava Thompson**, has asked for **RoomDesk**: a small web application where employees can find a room, check availability, book a meeting, and cancel it, and where she can manage the rooms themselves.

Room	Floor	Seats	Video	Notes
Harbor	1	4	No	Small huddle room
Lighthouse	1	8	Yes	The busiest room
Summit	2	12	Yes	Workshops
Focus Pod	2	2	No	One-to-ones
Atlas	3	20	Yes	Boardroom
Archive	3	6	No	Retired; not bookable

The rules are short to state and easy to get wrong:

- No two **confirmed** bookings of the same room may overlap. Back-to-back meetings (one ends at 10:30, the next starts at 10:30) are fine, and cancelled bookings never block a room.
- Meetings happen **Monday to Friday, 08:00–18:00** office time, start and end on **15-minute** boundaries, and last from 15 minutes to 4 hours.
- The number of attendees must fit the room's capacity, and retired rooms cannot be booked.
- Nobody can book in the past or more than **60 days** ahead.
- Cancelling marks a booking as cancelled; it is never deleted, and a meeting that has already started cannot be cancelled.

Five employees appear in the examples: Ava Thompson (Operations), Marcus Lee and Hannah Kim (Engineering), Priya Nair (Design), and Diego Santos (Sales). Their email addresses use the reserved `brightwave.example` domain, for example `ava@brightwave.example`, so no example can ever reach a real mailbox.

One detail makes every example in the book reproducible: RoomDesk's development configuration freezes the application's clock at **Monday, March 16, 2026, 07:30** office time. Whatever day you read this, "today" in RoomDesk is that Monday morning, so the rule "you cannot book in the past" gives the same answer on your machine as in the book. Chapter 4 shows how this works without a single `if (isDemo)` in the business code.

Creating the RoomDesk Solution

A **solution** groups related projects so that you can build, test, and open them together. RoomDesk has four projects at the start:

```
RoomDesk/
  RoomDesk.slnx
  src/
    RoomDesk.Core/     class library: entities, rules, DTOs (no ASP.NET, no EF Core)
    RoomDesk.Api/      ASP.NET Core Web API (minimal APIs)
    RoomDesk.Web/      Blazor Web App (Interactive Server)
  tests/
    RoomDesk.Tests/    xUnit tests
```

Separating `src` from `tests` keeps production code and test code apart. A second test project for Blazor components joins in Chapter 26.

Create a folder for the solution wherever you keep your code; the book uses `C:\src\RoomDesk` on Windows and `~/src/RoomDesk` elsewhere. Then create the solution file:

Terminal

```
$ mkdir RoomDesk
$ cd RoomDesk
$ dotnet new sln -n RoomDesk
The template "Solution File" was created successfully.
```

With the .NET 10 SDK, this creates `RoomDesk.slnx`, the newer XML-based solution format. The old `.sln` format was a dense, GUID-heavy text file that caused painful merge conflicts; `.slnx` is short, readable, and diff-friendly. Now create the four projects:

Terminal

```
$ dotnet new classlib -n RoomDesk.Core -o src/RoomDesk.Core
$ dotnet new webapi -n RoomDesk.Api -o src/RoomDesk.Api
$ dotnet new blazor -n RoomDesk.Web -o src/RoomDesk.Web --interactivity Server
$ dotnet new xunit -n RoomDesk.Tests -o tests/RoomDesk.Tests
```

Each command prints a confirmation and restores the project's NuGet packages, for example:

Terminal (excerpt)

```
The template "ASP.NET Core Web API" was created successfully.

Processing post-creation actions...
Restoring C:\src\RoomDesk\src\RoomDesk.Api\RoomDesk.Api.csproj:
  Determining projects to restore...
  Restored C:\src\RoomDesk\src\RoomDesk.Api\RoomDesk.Api.csproj (in 1.31 sec).
Restore succeeded.
```

The `-n` option names the project (and its root namespace and assembly), and `-o` chooses the output folder. The Blazor template's `--interactivity Server` option sets up Interactive Server rendering per page or component, which is the model RoomDesk uses: pages are static by default and opt in to interactivity where they need it.

Next, add the projects to the solution and wire up the references:

Terminal

```
$ dotnet sln add src/RoomDesk.Core src/RoomDesk.Api src/RoomDesk.Web
Project `src\RoomDesk.Core\RoomDesk.Core.csproj` added to the solution.
Project `src\RoomDesk.Api\RoomDesk.Api.csproj` added to the solution.
Project `src\RoomDesk.Web\RoomDesk.Web.csproj` added to the solution.

$ dotnet sln add tests/RoomDesk.Tests
Project `tests\RoomDesk.Tests\RoomDesk.Tests.csproj` added to the solution.

$ dotnet add src/RoomDesk.Api reference src/RoomDesk.Core
$ dotnet add src/RoomDesk.Web reference src/RoomDesk.Core
$ dotnet add tests/RoomDesk.Tests reference src/RoomDesk.Core
```

The CLI places each project in a solution folder that mirrors its directory. Open the solution file and you see how compact the new format is:

RoomDesk.slnx

```
<Solution>
  <Folder Name="/src/">
    <Project Path="src/RoomDesk.Api/RoomDesk.Api.csproj" />
    <Project Path="src/RoomDesk.Core/RoomDesk.Core.csproj" />
    <Project Path="src/RoomDesk.Web/RoomDesk.Web.csproj" />
  </Folder>
  <Folder Name="/tests/">
    <Project Path="tests/RoomDesk.Tests/RoomDesk.Tests.csproj" />
  </Folder>
</Solution>
```

The references define the direction of dependencies, and the direction is the architecture:

```
  RoomDesk.Web ----+
                   +----> RoomDesk.Core <---- RoomDesk.Tests
  RoomDesk.Api ----+                              |
```

```
        ^                                       |
        +-------------- (added in Chapter 26) -----+
```

`RoomDesk.Core` depends on nothing but .NET itself: no ASP.NET Core, no EF Core. That keeps the booking rules easy to test and lets both the API and the Blazor app share them. The Web project does *not* reference the API project; it talks to the API over HTTP, exactly as any other client would. That separation is what lets us deploy, scale, and secure the two applications independently later.

> **NOTE**
>
> On the .NET 8 SDK, `dotnet new sln` creates a classic `RoomDesk.sln` file. Everything in this book works the same with it; only the file name differs in commands. The .NET 10 SDK can still create the old format with `dotnet new sln --format sln` if your tools require it.

Finally, delete the placeholder `Class1.cs` in `src/RoomDesk.Core`; the first real file for that project arrives in Chapter 2. Build and test everything from the solution folder:

Terminal

```
$ dotnet build
  RoomDesk.Core -> C:\src\RoomDesk\src\RoomDesk.Core\bin\Debug\net10.0\RoomDesk.Core.dll
  RoomDesk.Tests -> C:\src\RoomDesk\tests\RoomDesk.Tests\bin\Debug\...\RoomDesk.Tests.dll
  RoomDesk.Api -> C:\src\RoomDesk\src\RoomDesk.Api\bin\Debug\net10.0\RoomDesk.Api.dll
  RoomDesk.Web -> C:\src\RoomDesk\src\RoomDesk.Web\bin\Debug\net10.0\RoomDesk.Web.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

$ dotnet test
Passed!  - Failed:     0, Passed:     1, Skipped:     0, Total:     1, Duration: 8 ms
         - RoomDesk.Tests.dll (net10.0)
```

When a folder contains exactly one solution or project file, `dotnet build` and `dotnet test` find it automatically. The single passing test is the template's empty placeholder.

> **TIP**
>
> Commit now. Run `dotnet new gitignore` in the solution folder to add a `.gitignore` that excludes `bin/`, `obj/`, editor folders, and other build output, then `git init` and make your first commit. Small, frequent commits make it painless to compare your code with the book when something does not work.

Running the Template Apps

Before we change anything, run the generated applications to make sure the toolchain works. Start the API:

Terminal

```
$ dotnet run --project src/RoomDesk.Api
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5080
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\src\RoomDesk\src\RoomDesk.Api
```

Your first run shows a different port number, because the template picks random ports to avoid collisions between projects. We fix that in a moment. In a second terminal, call the sample endpoint:

Terminal

```
$ curl http://localhost:5080/weatherforecast
[{"date":"2026-10-04","temperatureC":20,"summary":"Chilly","temperatureF":67},
 {"date":"2026-10-05","temperatureC":28,"summary":"Warm","temperatureF":82},
 ...]
```

The values are random, so yours will differ. Notice two things about the JSON: property names are camelCase (`temperatureC`, although the C# property is `TemperatureC`), and the `DateOnly` value is serialized as an ISO 8601 date, `"2026-10-04"`. Those are the defaults of `System.Text.Json` in ASP.NET Core, and we rely on them throughout the book. On Windows PowerShell, type `curl.exe` instead of `curl`; Chapter 5 explains why.

The API template also generates an **OpenAPI** description of the API in Development. Browse to `http://localhost:5080/openapi/v1.json` and you see a JSON document that starts like this:

GET /openapi/v1.json (excerpt)

```
{
  "openapi": "3.1.1",
  "info": {
    "title": "RoomDesk.Api | v1",
    "version": "1.0.0"
  },
  ...
```

Chapter 14 builds on this document and adds an interactive UI to explore the API. Stop the API with **Ctrl+C**, then run the Blazor app:

Terminal

```
$ dotnet run --project src/RoomDesk.Web
```

The Web template's launch profile opens a browser automatically. You see a page with a dark sidebar (Home, Counter, Weather) and the heading "Hello, world!". Click **Counter** and press the button a few times. The count updates without a page reload, yet there is no JavaScript of yours anywhere: the `Counter` component runs on the server in Interactive Server mode, and Blazor's SignalR connection sends the click to the server and the changed number back. The **Weather** page, in contrast, is rendered with static SSR plus streaming. We take these pages apart in Chapter 16.

Fixing the ports

Random ports are inconvenient in a book and in a team: the Web app needs to know where the API lives. Each web project has a `Properties/launchSettings.json` file with launch **profiles** used by `dotnet run` and by editors. Change the generated port numbers so that the API uses 5080 (HTTP) and 7080 (HTTPS):

src/RoomDesk.Api/Properties/launchSettings.json

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "http://localhost:5080",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "https://localhost:7080;http://localhost:5080",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Do the same in `src/RoomDesk.Web/Properties/launchSettings.json` with **5090** and **7090**. These are the addresses used everywhere in this book:

Application	HTTP	HTTPS
RoomDesk.Api	`http://localhost:5080`	`https://localhost:7080`
RoomDesk.Web	`http://localhost:5090`	`https://localhost:7090`