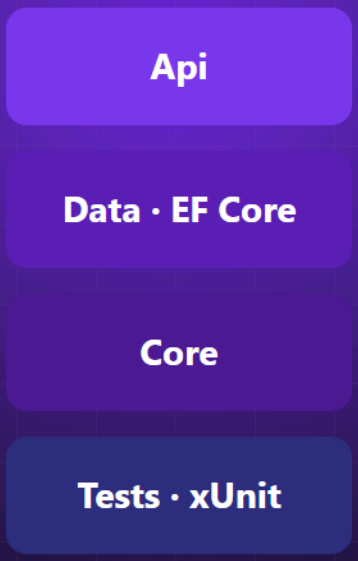
PricingService.cs

```
public sealed record Money(
  decimal Amount, string Currency);

public Money PriceFor(Book book,
  Customer? customer) =>
  customer is { LoyaltyPoints: >= 500 }
    ? book.Price with {
      Amount = book.Price.Amount * 0.95m }
    : book.Price;

// records · patterns · nullable · LINQ
```




```
$ dotnet test Passed!
$ dotnet publish -c Release
$ curl localhost:8080/api/books/1
```

A COMPLETE, PRACTICAL GUIDE

C# and .NET from Zero to Production

Learn modern C# and build, test, and ship a real web API with .NET

C# 14 · .NET 10 · LINQ · async/await · xUnit
ASP.NET Core · EF Core · Native AOT · Containers

LANGUAGE · LIBRARIES · WEB · PRODUCTION

THENEXBOOKS

C# and .NET from Zero to Production

A Complete, Practical Guide to Modern C# and .NET
From your first program to a tested, deployed web API
— for C# 14 and .NET 10

First Edition

THENEXBOOKS

C# and .NET from Zero to Production
A Complete, Practical Guide to Modern C# and .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, C#, and the libraries discussed in this book evolve; always consult the official documentation for the versions you use.

Microsoft, .NET, C#, Visual Studio, Visual Studio Code, Azure, and SQL Server are trademarks of the Microsoft group of companies. JetBrains and Rider are trademarks of JetBrains s.r.o. Docker is a trademark of Docker, Inc. PostgreSQL is a trademark of the PostgreSQL Community Association of Canada. 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 or the .NET Foundation.

Quillfield Books, the business used as the running example in this book, is fictional, as are its shops, customers, books, and data. ISBN-like values in examples are sample data.

Preface

C# is a remarkably versatile language. The same language and the same platform let you write a quick command-line tool, a cross-platform desktop or mobile app, a game, a cloud service handling heavy traffic, or a background worker crunching data — on Windows, macOS, or Linux. Modern .NET is open source, fast, and evolves every year, and modern C# is expressive, safe, and pleasant to write.

That breadth is also what makes C# intimidating at first. There are classes and records, structs and interfaces, LINQ and async, dependency injection and configuration, Minimal APIs and Entity Framework Core — and every tutorial seems to assume you already know half of it. This book takes a different approach. We start from zero and build one real application step by step, introducing each idea at the moment it solves a problem we actually have.

By the end, that application has grown from a small console program into a tested, documented web API backed by a database, packaged into a container, and ready for production. Along the way you will learn not only how to write C#, but how professional .NET developers structure, test, and ship their code — and *why*.

Who This Book Is For

- **Beginners** who want to learn programming with a modern, well-designed language, and are ready to practice.
- **Developers from other languages** such as Java, Python, JavaScript, or Go who need to become productive in C# and .NET.
- **C# developers returning after a few years** who want to catch up with records, pattern matching, nullable reference types, Minimal APIs, and the latest language features.
- **Students and career changers** who want a portfolio-quality project built the way professional teams build software.

What You Will Learn

- How .NET works, how to set up your tools, and how to create, build, run, and debug projects.
- The C# language from the ground up: types, strings, control flow, methods, classes, interfaces, structs, records, and enums.
- Modern C#: nullable reference types, pattern matching, collection expressions, primary constructors, and C# 14 features.
- Collections, generics, delegates, lambdas, LINQ, exceptions, files, JSON, and dates and times.
- Asynchronous programming with async and await, cancellation, parallelism, and channels.
- Dependency injection, configuration, logging, and testing with xUnit.
- Web APIs with ASP.NET Core Minimal APIs, Entity Framework Core, validation, OpenAPI, authentication, and background services.

- Performance, Native AOT, containers, health checks, OpenTelemetry, and CI/CD.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) sets up .NET and covers the language fundamentals. **Part II — Object-Oriented and Type-Safe C#** (Chapters 6–10) covers classes, interfaces, structs, records, null safety, and pattern matching. **Part III — Core Libraries** (Chapters 11–15) covers collections, generics, LINQ, error handling, files, JSON, and time. **Part IV — Asynchronous and Professional .NET** (Chapters 16–19) covers async programming, concurrency, the generic host, and testing. **Part V — Building Web Applications** (Chapters 20–24) builds the Quillfield web API with data access, validation, security, and background work. **Part VI — Production** (Chapters 25–28) covers performance, deployment, observability, and a complete case study. The appendices provide a cheat sheet, templates, a production checklist, and a glossary.

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

The Running Example: Quillfield Books

Throughout the book we build the software for **Quillfield Books**, a fictional chain of three independent bookshops — Riverside, Old Town, and Harbor — that also sells online. It starts with a simple need: know which books are in stock where. Over the chapters, a console tool grows to import catalogs, track stock and orders, apply loyalty discounts, and produce sales reports. In Part V it becomes a web API with a real database, validation, authentication, and background jobs, and in Part VI we take it to production.

Versions and Conventions

Examples target **.NET 10**, a long-term support release, and **C# 14**. Most code also runs on .NET 8 and .NET 9; where a feature needs a newer language or runtime version, we say so. The ecosystem moves quickly, so where details may change between releases, a Note points you to the official documentation rather than guessing.

> **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.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `src/Quillfield.Core/Book.cs`, tell you where the code lives. Terminal commands work the same on Windows, macOS, and Linux unless noted.

Contents

Preface

PART I — GETTING STARTED

1 Why C# and .NET? The Platform, the Language, and Your Setup

2 Your First Programs: Top-Level Statements, Projects, and File-Based Apps

3 Types, Variables, and Operators

4 Strings, Formatting, and Console Input and Output

5 Control Flow and Methods

PART II — OBJECT-ORIENTED AND TYPE-SAFE C#

6 Classes, Objects, and Properties

7 Inheritance, Interfaces, and Polymorphism

8 Structs, Records, Enums, and Value Semantics

9 Nullable Reference Types and Null Safety

10 Pattern Matching and Modern Data Modeling

PART III — CORE LIBRARIES

11 Collections: Arrays, Lists, Dictionaries, Sets, and Collection Expressions

12 Generics, Delegates, Lambdas, and Events

13 LINQ in Depth

14 Exceptions, Error Handling, and Result Types

15 Files, Streams, JSON, and Dates and Times

PART IV — ASYNCHRONOUS AND PROFESSIONAL .NET

16 async and await from First Principles

17 Tasks, Cancellation, Parallelism, and Channels

18 Dependency Injection, Configuration, and Logging with the Generic Host

19 Testing with xUnit: Unit Tests, Test Doubles, and Good Test Design

PART V — BUILDING WEB APPLICATIONS

20 ASP.NET Core Minimal APIs

21 Data Access with Entity Framework Core

22 Validation, Error Handling, and OpenAPI for Web APIs

23 Authentication and Authorization in ASP.NET Core

24 Calling Other Services with HttpClient and Running Background Work

PART VI — PRODUCTION

25 Performance: Memory, Span<T>, Benchmarking, and the Garbage Collector

26 Packaging and Deployment: dotnet publish, Native AOT, and Containers

27 Observability and Production Readiness: Health Checks, OpenTelemetry, and CI/CD

28 Case Study: Quillfield Books from Console App to Cloud API

APPENDICES

A C# Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Getting Started

What .NET is and how to set it up, your first programs and projects, and the fundamentals every C# program is built from: types, operators, strings, console input and output, control flow, and methods.

1 · Why C# and .NET?
2 · Your First Programs
3 · Types, Variables, and Operators
4 · Strings and Console I/O
5 · Control Flow and Methods

CHAPTER 1 · PART I — GETTING STARTED

Why C# and .NET? The Platform, the Language, and Your Setup

Before we write a single line of code, we need a clear picture of what C# and .NET actually are, how the pieces fit together, and how to get a working development environment on your machine. By the end of this chapter you will have the .NET 10 SDK installed, an editor ready, and a first look at Quillfield Books, the project we will grow throughout this book.

What You Are Getting Into

Learning a programming language is a bit like learning to cook in a new kitchen. The recipes (the language) matter, but so do the stove, the knives, and the pantry (the platform and its tools). Many beginners struggle not because the language is hard, but because nobody explained the kitchen. They see words like *runtime*, *SDK*, *framework*, and *assembly* thrown around and assume everyone else already knows what they mean.

This chapter is our tour of the kitchen. We will explain each piece from first principles, without assuming you have written C# before. If you come from Java, Python, or JavaScript, many ideas will feel familiar, and we will point out the similarities and the differences as we go. If you are completely new to programming, that is fine too: every term is defined the first time it appears.

What Is C#?

C# (pronounced "see sharp") is a general-purpose programming language created by Microsoft and now developed in the open, with its design discussions and proposals published publicly. "General-purpose" means it is not tied to one kind of program. People use C# to build web APIs and websites, desktop applications, command-line tools, cloud services, games (the Unity game engine uses C# for scripting), mobile apps, and background workers that process data around the clock.

C# has a few defining characteristics that are worth understanding early, because they shape how you will write code:

- **Statically typed.** Every variable has a type that the compiler knows before the program runs. If you try to put text into a variable meant for a number, the compiler stops you with an error. This catches a large class of mistakes before your program ever reaches a user.
- **Object-oriented, with functional features.** You can model your world with classes and objects (a `Book`, a `Shop`, an `Order`), and you can also write concise, expression-based code with lambdas, LINQ queries, and pattern matching.

- **Memory-managed.** You do not manually allocate and free memory as you would in C. A component called the *garbage collector* cleans up objects you no longer use.
- **Compiled.** Your source code is translated by a compiler before it runs. As we will see shortly, C# actually goes through two translation steps.
- **Evolving steadily.** A new version of the language ships with each yearly .NET release, adding features that make code shorter, safer, or faster.

A one-paragraph history

C# first appeared around the turn of the millennium as part of Microsoft's original .NET Framework, a Windows-only platform. For many years, C# and .NET were strongly associated with Windows. That changed with the arrival of .NET Core, a rewritten, open-source, cross-platform version of .NET that runs on Windows, macOS, and Linux. Over time, .NET Core became simply ".NET", dropping the "Core" suffix starting with .NET 5, and the old Windows-only .NET Framework moved into maintenance mode. Meanwhile the language kept growing: generics, LINQ, `async` and `await`, records, pattern matching, nullable reference types, and many more features arrived release by release. Today, with .NET 10 and C# 14, you get a modern, open-source, cross-platform language and runtime that is used for everything from small scripts to very large cloud systems.

Why choose C#?

There is no single "best" language, and we will not pretend otherwise. Python is wonderful for data science and quick scripts, JavaScript is unavoidable in the browser, and Go and Rust have their own strengths. Here is why many teams pick C# and .NET, and why we think it is an excellent language to learn deeply:

- **One language, many targets.** The same skills apply to web APIs, console tools, background services, and desktop apps.
- **Excellent tooling.** The compiler gives precise error messages, editors offer strong autocompletion and refactoring, and the debugger is first-class.
- **A huge standard library.** Collections, file I/O, JSON, HTTP, cryptography, dates and times, and much more come built in, so you write less glue code.
- **Performance.** .NET has received sustained performance investment with each release. You can write high-level, readable code and still get efficient execution, and when you need to go further, tools like `Span<T>` (Chapter 25) let you control memory precisely.
- **Safety.** Static typing, nullable reference types (Chapter 9), and memory management remove whole categories of bugs.
- **Long-term support.** Predictable releases with clearly defined support windows make it practical for businesses to plan.

> **NOTE**
>
> If you know Java, C# will feel very familiar: classes, interfaces, garbage collection, and a curly-brace syntax. The differences you will notice first are properties (instead of getter and setter methods), `async` / `await` built into the language, LINQ, value types you can define yourself (`struct`), and records. If you know Python or JavaScript, the biggest shift is static typing: you declare (or let the compiler infer) types, and the compiler checks them before the program runs.

What Is .NET?

People often use "C#" and ".NET" interchangeably, but they are different things. **C#** is a language: a set of rules for writing programs. **.NET** is the platform that runs those programs and the toolkit you use to build them. Other languages run on .NET too, notably F# (a functional-first language) and Visual Basic, but C# is by far the most widely used of them.

The .NET platform consists of several cooperating pieces. Let us look at each one.

The runtime and the CLR

The **runtime** is the software that actually executes your program. Its core is called the **CLR**, the Common Language Runtime. The CLR is responsible for:

- Loading your compiled program into memory.
- Translating it into machine code your processor understands (more on this below).
- Managing memory, including running the garbage collector.
- Enforcing type safety, so code cannot treat a piece of memory as something it is not.
- Handling exceptions, threads, and many other low-level concerns.

You do not interact with the CLR directly very often, but it is always there, doing work on your behalf. Whenever you hear "managed code", it means code that runs under the supervision of the CLR.

The base class library

The **base class library** (BCL) is the large set of ready-made types that ships with .NET. When you write `Console.WriteLine`, you are calling a method on the `Console` class from the BCL. When you use `List<T>` to store a list of books, `File.ReadAllText` to read a file, or `JsonSerializer` to produce JSON, those all come from the BCL. The BCL is organized into **namespaces**, which are like folders for types: `System`, `System.Collections.Generic`, `System.IO`, `System.Text.Json`, and so on.

Beyond the BCL there are larger frameworks built on top of it. **ASP.NET Core** is the framework for web applications and APIs, which we will use in Part V. **Entity Framework Core** (EF Core) is a data-access library for working with databases, which we cover in Chapter 21. These are also part of the .NET family, but you add them to a project when you need them.

The SDK and the dotnet command

The **SDK** (Software Development Kit) is what developers install. It contains the compiler, the build system, project templates, and the command-line tool called `dotnet`. The SDK also includes a runtime, so once you install the SDK you can both build and run programs.

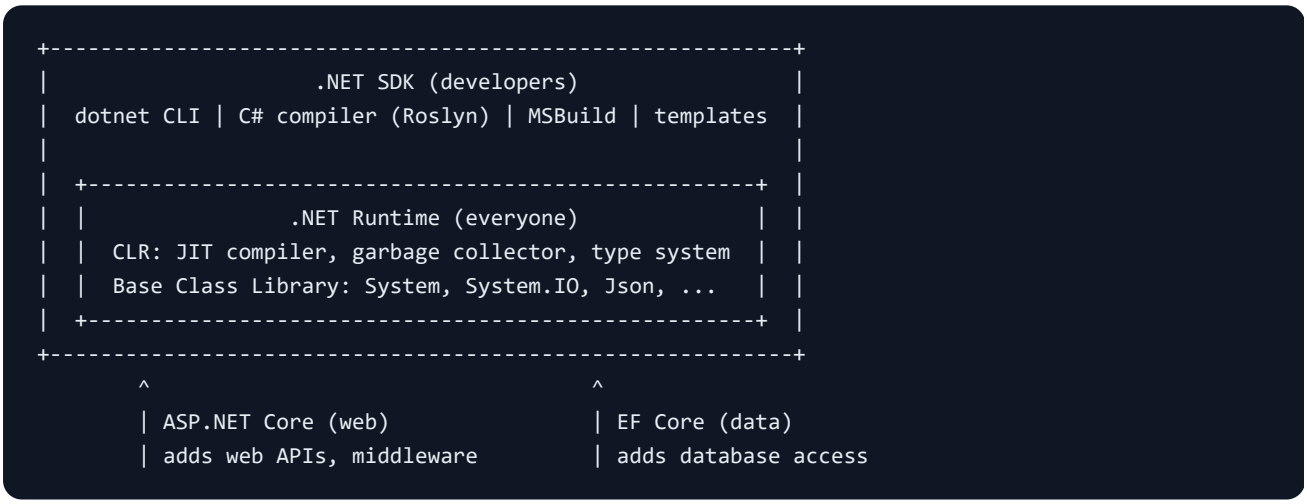
The `dotnet` command is your main interface to the platform. You will use it constantly:

- `dotnet new` creates a new project from a template.
- `dotnet build` compiles a project.
- `dotnet run` builds and runs a project in one step.
- `dotnet test` runs automated tests.
- `dotnet add package` adds a library from NuGet, the .NET package repository.
- `dotnet publish` prepares an application for deployment.

End users of your application, such as the server that will run the Quillfield API in production, only need the **runtime**, not the full SDK. In fact, as Chapter 26 shows, you can even package the runtime together with your application so that the target machine needs nothing preinstalled.

Here is how the pieces relate:

Diagram: the .NET platform



How Your Code Runs: Compilation, IL, JIT, and AOT

Computers do not understand C#. A processor understands only *machine code*: very low-level instructions specific to a processor family such as x64 or Arm64. Something has to translate your C# into machine code. In .NET this happens in two stages, and understanding them will demystify a lot of what you see later.

Stage one: C# to IL

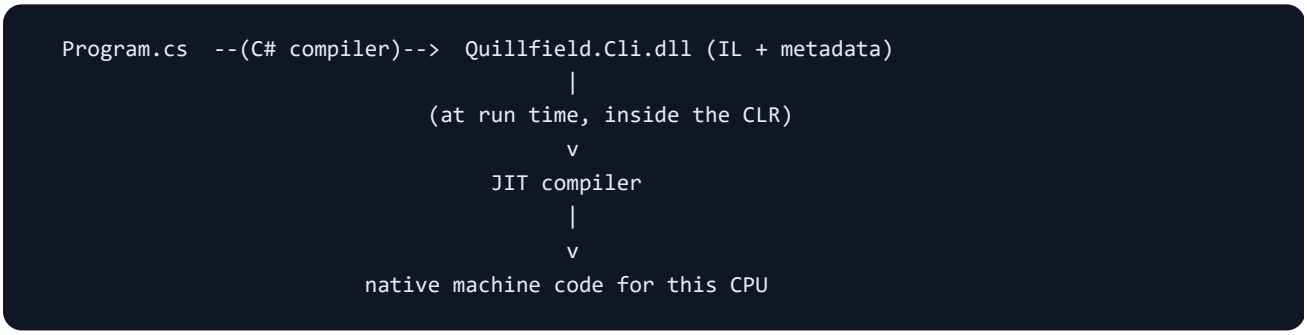
When you run `dotnet build`, the C# compiler (whose open-source code name is **Roslyn**) reads your `.cs` files and checks them for errors. If everything is valid, it produces an **assembly**: a `.dll` file containing **IL** (Intermediate Language) plus **metadata** describing every type and method. IL is a processor-independent instruction set. It is lower level than C#, but it is not tied to any particular CPU.

Because the compiler checks your code at this stage, many mistakes are caught before anything runs. Misspell a method name, pass the wrong type of argument, or forget a semicolon, and the build fails with an error message that points to the exact file and line.

Stage two: IL to machine code with the JIT

When you run the program, the CLR loads the assembly. The first time a method is called, the **JIT compiler** (Just-In-Time compiler) translates that method's IL into machine code for the current processor, then runs it. Subsequent calls reuse the compiled machine code. The JIT can take advantage of the exact processor it is running on, and modern .NET uses techniques such as *tiered compilation*: it compiles methods quickly at first, then recompiles frequently used ("hot") methods with heavier optimizations once it sees they matter.

Diagram: the default compile-and-run pipeline



This two-stage design is why the same `.dll` can run on Windows, macOS, and Linux, on Intel and Arm processors alike: the IL is portable, and each machine's runtime does the final translation.

Ahead-of-time compilation (Native AOT)

There is an alternative. With **Native AOT** (Ahead-Of-Time compilation), the entire application is compiled to machine code at publish time, producing a self-contained native executable. There is no JIT at run time. The benefits are fast startup and a smaller memory footprint, which matters for command-line tools, serverless functions, and containers that scale up and down frequently. The trade-off is that some dynamic features, especially code that relies heavily on reflection (inspecting and creating types at run time), are restricted or need extra configuration. We cover Native AOT and its limits properly in Chapter 26. For now, it is enough to know that both models exist and that JIT is the default.

> **TIP**
>
> You do not need to choose between JIT and AOT while learning. Write normal C#, run it with `dotnet run`, and let the JIT do its job. Native AOT is a deployment decision you can make later, once your application works.

Versions, Releases, and Support: LTS and STS

.NET ships a new major version every year, in November. Each release is one of two kinds:

Release type	Meaning	Examples
LTS (Long-Term Support)	Supported for a longer period (typically three years). The safe choice for production systems.	.NET 8, .NET 10
STS (Standard-Term Support)	Supported for a shorter period. Good for trying new features early.	.NET 9

Even-numbered releases have been LTS and odd-numbered releases STS in recent years. Support windows are periodically reviewed by Microsoft, so always check the official .NET support policy page for current end-of-support dates rather than relying on memory. During its support period, a release receives regular servicing updates with security and bug fixes. These are "patch" versions, such as 10.0.1, 10.0.2, and so on, and you should keep up with them.

Each .NET release also ships with a matching C# version:

.NET version	Release type	Default C# version
.NET 8	LTS	C# 12
.NET 9	STS	C# 13
.NET 10	LTS	C# 14

This book targets **.NET 10 (LTS) with C# 14**, released in November 2025. It is the version we recommend for any new project today. Most of the code in this book also works on .NET 8 and .NET 9. When we use a feature that requires C# 13 or C# 14, or an API added in .NET 9 or .NET 10, we will say so explicitly, so you can adapt if your workplace is still on an older version.

> **NOTE**
>
> The C# language version is chosen by your project's *target framework*. A project that targets `net10.0` uses C# 14 by default. You can override the language version in the project file, but mixing an old framework with a newer language version is unsupported and can lead to confusing errors. Keep the defaults unless you have a specific reason.

Installing the .NET 10 SDK

Now let us get the tools onto your machine. You need the **.NET 10 SDK**. The official download page is on the Microsoft .NET website (search for "download .NET" and use the dotnet.microsoft.com page). Always download from official sources, and choose the SDK, not just the runtime, because you are a developer.

Windows

On Windows you have several good options:

- **Installer.** Download the .NET 10 SDK installer for your architecture (x64 for most PCs, Arm64 for Arm-based laptops) and run it.
- **Windows Package Manager.** If you use `winget`, you can install the SDK from the command line. Package IDs occasionally change, so search first.
- **Visual Studio.** Installing Visual Studio with the ".NET desktop development" or "ASP.NET and web development" workload also installs a .NET SDK.

Terminal (Windows, PowerShell)

```
PS> winget search Microsoft.DotNet.SDK
PS> winget install Microsoft.DotNet.SDK.10
```

macOS

On macOS, download the installer package (`.pkg`) for your processor. Apple Silicon Macs (M-series chips) need the Arm64 installer; older Intel Macs need the x64 installer. Some developers prefer package managers such as Homebrew; if you go that route, make sure the package you install provides the .NET 10 SDK.

Linux

Most popular Linux distributions offer .NET through their own package repositories or through Microsoft's repository. Package names vary by distribution, and some distributions publish new versions on their own schedule. Microsoft also provides an install script (`dotnet-install.sh`) that installs the SDK into your home directory without administrator rights, which is handy on machines you do not control. Check the official Linux installation documentation for your distribution.

Terminal (Ubuntu example, check docs for your distribution)

```
$ sudo apt-get update
$ sudo apt-get install -y dotnet-sdk-10.0
```

> **WARNING**
>
> Do not download .NET installers or "portable SDK" bundles from random websites or file-sharing links. The SDK runs code on your machine with your permissions. Use the official .NET website, your operating system's package manager, or Visual Studio's installer. The same rule applies later when you add NuGet packages: prefer well-known, actively maintained packages.

Verifying the Installation with dotnet --info

After installing, open a *new* terminal window (so that it picks up the updated PATH environment variable) and run:

Terminal

```
$ dotnet --version
10.0.100
```

The exact number depends on the SDK feature band and patch level you installed, but it should start with `10.0`. For a more complete picture, run `dotnet --info`:

Terminal (output trimmed; your values will differ)

```
$ dotnet --info
.NET SDK:
 Version:           10.0.100
 Commit:            ...
 Workload version:  10.0.100-manifests...

Runtime Environment:
 OS Name:     Windows
 OS Version:  10.0.22631
 OS Platform: Windows
 RID:         win-x64
 Base Path:   C:\Program Files\dotnet\sdk\10.0.100\

.NET SDKs installed:
  8.0.400 [C:\Program Files\dotnet\sdk]
  10.0.100 [C:\Program Files\dotnet\sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.App 10.0.0 [C:\Program Files\dotnet\shared\...]
  Microsoft.NETCore.App 10.0.0 [C:\Program Files\dotnet\shared\...]
```

Let us read this output piece by piece, because it answers questions you will have later:

- **.NET SDK Version** is the SDK that the `dotnet` command will use in the current folder.
- **RID** is the *Runtime Identifier*, such as `win-x64`, `linux-x64`, or `osx-arm64`. It describes your operating system and processor. RIDs matter when you publish self-contained apps in Chapter 26.

- **.NET SDKs installed** lists every SDK on the machine. Multiple SDKs can live side by side without conflict.
- **.NET runtimes installed** lists the runtimes. `Microsoft.NETCore.App` is the base runtime; `Microsoft.AspNetCore.App` adds what ASP.NET Core applications need.

> **TIP**
> Side-by-side installs are normal and healthy. If a project needs a specific SDK version, you can pin it with a `global.json` file in the project folder. Run `dotnet new globaljson --sdk-version 10.0.100` to create one, and adjust the version to match what `dotnet --list-sdks` shows on your machine.

A thirty-second smoke test

Let us make sure everything actually works by creating and running a tiny program. We will explain every part of this in Chapter 2; for now, just type the commands:

Terminal

```
$ dotnet new console -o HelloQuillfield
The template "Console App" was created successfully.

$ cd HelloQuillfield
$ dotnet run
Hello, World!
```

If you see `Hello, World!`, your SDK is installed correctly, the compiler works, and the runtime can execute your program. The first run may take a few seconds longer than later runs, because the SDK sets up caches the first time.

Choosing an Editor

You can write C# in any text editor, but a good editor with C# support makes an enormous difference, especially when you are learning. It shows errors as you type, suggests completions, explains types when you hover over them, and lets you step through code with a debugger. Three popular choices are listed below. All of them are capable; pick the one that suits your operating system, budget, and taste.

Editor	Platforms	Notes
Visual Studio	Windows	Full IDE from Microsoft. Community edition is free for individuals and many small teams (check the license terms).
Visual Studio Code with C# Dev Kit	Windows, macOS, Linux	Lightweight, free editor. Install the C# Dev Kit extension for project management, testing, and debugging (check its license).
JetBrains Rider	Windows, macOS, Linux	Cross-platform IDE with strong refactoring. Commercial, with free options for some uses (check current terms).

Whichever you choose, this book uses the `dotnet` command line for creating, building, and running projects. That keeps the instructions identical for everyone, and the commands work the same way inside any editor's built-in terminal. Editors call the same SDK under the hood, so anything you do with buttons in an IDE can also be done from the terminal, and vice versa.

Meet Quillfield Books

Small examples are useful for explaining one feature at a time, but real skill comes from building something that grows. Throughout this book we will build software for **Quillfield Books**, a fictional small chain of independent bookshops. Quillfield has three shops and also takes online orders:

Code	Shop name	Notes
RIV	Riverside	The original shop, largest stock.
OLD	Old Town	Small shop with many children's books.
HRB	Harbor	Newest shop, near the ferry terminal.

Quillfield's problems are the everyday problems of a small business. Staff need to look up which books are in stock at which shop. Someone needs to notice when a popular title is running low so it can be reordered. Loyal customers earn points and get a discount once they have enough. Online orders need to be recorded, priced correctly, and tracked from *Pending* to *Delivered*. And at the end of each month, the owners want a sales report.

These problems are simple enough to understand immediately, yet rich enough to exercise nearly every feature of C# and .NET. Here are a few sample books from the catalog we will use (titles, authors, and ISBN-style numbers are all invented sample data):

Title	Genre	Price
The Lantern Keeper	Fiction	$18.99
Murder at Millbrook	Mystery	$14.50
Orbit of Ash	ScienceFiction	$21.00
Bread & Salt	Cookery	$32.00
The Little Ferry	Children	$9.99

Our customers include Ava Thompson, Noah Garcia, and Mia Rossi, all with `@example.com` email addresses (a domain reserved for documentation, so no real person receives mail there).

How Quillfield grows through the book

The software evolves in the same way real projects do, one capability at a time:

1. **Parts I and II**: a console application, `Quillfield.Cli`, and a class library, `Quillfield.Core`, holding the domain types such as `Book`, `Shop`, `Customer`, `Order`, `Isbn`, and `Money`.
2. **Part III**: collections, LINQ, error handling, CSV import, and JSON, so the CLI can import a catalog and produce reports.
3. **Part IV**: asynchronous code, dependency injection, configuration, logging, and automated tests with xUnit.
4. **Part V**: an ASP.NET Core Minimal API, `Quillfield.Api`, backed by a database through EF Core in `Quillfield.Data`.
5. **Part VI**: performance, packaging, containers, health checks, observability, and continuous integration, ready for production.

By the end, the solution will look like this:

Diagram: the final Quillfield solution layout

```
Quillfield.sln
|-- src/
|   |-- Quillfield.Core/        domain types and services (no I/O)
|   |-- Quillfield.Cli/         console app: books, stock, import, reports
|   |-- Quillfield.Data/        EF Core QuillfieldDbContext and migrations
|   `-- Quillfield.Api/         ASP.NET Core Minimal API under /api
`-- tests/
    |-- Quillfield.Core.Tests/  xUnit unit tests
    `-- Quillfield.Api.Tests/   xUnit integration tests
```

And here is a preview of the kind of command-line experience we will build in the first half of the book. Do not try to run it yet; it does not exist until later chapters:

Terminal (preview of later chapters)

```
$ quillfield books list --genre Mystery
Id  Title                  Author           Price
--  ---------------------  ---------------  -------
 2  Murder at Millbrook    James Okafor     $14.50

$ quillfield stock low --shop RIV
Low stock at Riverside (RIV):
  Orbit of Ash           qty 2   (reorder at 5)
  The Lantern Keeper     qty 3   (reorder at 5)
```

Notice that we will design the core logic (pricing, stock rules, order validation) to be independent of how it is used. The same `Quillfield.Core` code that powers the console tool will later power the web API. This separation is one of the most valuable habits in professional software development, and we will practice it from the start.